VISIT…

Making Everything Easier!™

3rd Edition

Beginning Programming with Java® FOR DUMMIES®

Learn to:

- Use basic development concepts and techniques with Java
- Debug Java programs and make them work
- Overcome standard programming challenges
- Work with all the latest features of Java 7

Barry Burd, PhD
Author of Java For Dummies

Get More and Do More at Dummies.com®

Start with **FREE** Cheat Sheets

Cheat Sheets include

- Checklists
- Charts
- Common Instructions
- And Other Good Stuff!

To access the Cheat Sheet created specifically for this book, go to
www.dummies.com/cheatsheet/beginningprogrammingwithjava

Get Smart at Dummies.com

Dummies.com makes your life easier with 1,000s of answers on everything from removing wallpaper to using the latest version of Windows.

Check out our

- Videos
- Illustrated Articles
- Step-by-Step Instructions

Plus, each month you can win valuable prizes by entering our Dummies.com sweepstakes. *

Want a weekly dose of Dummies? Sign up for Newsletters on

- Digital Photography
- Microsoft Windows & Office
- Personal Finance & Investing
- Health & Wellness
- Computing, iPods & Cell Phones
- eBay
- Internet
- Food, Home & Garden

Find out "HOW" at Dummies.com

**Sweepstakes not currently available in all countries; visit Dummies.com for official rules.*

Beginning Programming with Java®

FOR

DUMMIES®

3RD EDITION

Beginning Programming with Java®

FOR DUMMIES®

3RD EDITION

by Barry Burd
Author of *Java 2 For Dummies*

John Wiley & Sons, Inc.

Beginning Programming with Java® For Dummies®, 3rd Edition
Published by
John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030-5774

www.wiley.com

Copyright © 2012 by John Wiley & Sons, Inc., Hoboken, New Jersey

Published by John Wiley & Sons, Inc., Hoboken, New Jersey

Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permissions.

Trademarks: Wiley, the Wiley logo, For Dummies, the Dummies Man logo, A Reference for the Rest of Us!, The Dummies Way, Dummies Daily, The Fun and Easy Way, Dummies.com, Making Everything Easier, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and other countries, and may not be used without written permission. Java is a registered trademark of Oracle America, Inc. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002.

For technical support, please visit www.wiley.com/techsupport.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at http://booksupport.wiley.com. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2012934909

ISBN 978-0-470-37174-9 (pbk); ISBN 978-1-118-22014-6 (ePDF); ISBN 978-1-118-23384-9 (ePub); ISBN 978-1-118-25852-1 (eMobi)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

About the Author

Dr. Barry Burd has an M.S. in Computer Science from Rutgers University and a Ph.D. in Mathematics from the University of Illinois. As a teaching assistant in Champaign-Urbana, Illinois, he was elected five times to the university-wide List of Teachers Ranked as Excellent by their Students.

Since 1980, Dr. Burd has been a professor in the Department of Mathematics and Computer Science at Drew University in Madison, New Jersey. When he's not lecturing at Drew University, Dr. Burd leads training courses for professional programmers in business and industry. He has lectured at conferences in the United States, Europe, Australia, and Asia. He is the author of several articles and books, including *Java For Dummies* and *Android Application Development All-in-One For Dummies,* both published by John Wiley & Sons, Inc.

Dr. Burd lives in Madison, New Jersey with his wife and two kids (both in their twenties, and mostly on their own). As an avid indoor enthusiast, Dr. Burd enjoys sleeping, talking, and eating.

Dedication

For Harriet, Sam and Jennie, Sam and Ruth, Abram and Katie, Benjamin and Jennie

Author's Acknowledgments

Author's To-Do List, February 13, 2012:

Item: Send chocolate to Kelly Ewing — the book's project editor and copy editor. As anyone who reads Chapter 4 learns, chocolate is one of the most precious commodities on earth. So when I give chocolate, I give it thoughtfully and intentionally.

Item: Have a plaque erected in honor of Katie Feltman, your acquisitions editor at Wiley. While you worked on other projects, Katie kept on insisting that you write this book's third edition. (Sure, you wanted a long vacation instead of another book project, but who cares? She was right; you were wrong.)

Item: Send a thank-you note to tech editor John Mueller who helped polish your original work and, miraculously, didn't make a lot of extra work for you.

Item: Recommend your agent Neil Salkind to other computer book authors. If it weren't for Neil, you'd still be roaming the book exhibits and looking needy at the technology conferences.

Item: Visit Frank Thornton, Bonnie Averbach, and Herbert Putz at Temple University. Thank them for steering you to a career as a professor. In any other career, you'd have no time left to write. (And by the way, while you're in Philly, don't forget to stop for a cheesesteak.)

Item: Send e-mail to Gaisi Takeuti at the University of Illinois, and to William Wisdom and Hughes LeBlanc at Temple University. Thank them for teaching you about Symbolic Logic. It's made your life as a computer scientist and mathematician much richer.

Item: Spend more time with your family. (Remind them that you're the guy who wandered around the house before you started writing books.) Renew your pledge to clean up after yourself. Don't be so high-strung and finish each sentence that you start. Remember that you can never fully return the love they've given you, but you should always keep trying.

Publisher's Acknowledgments

We're proud of this book; please send us your comments at http://dummies.custhelp.com. For other comments, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002.

Some of the people who helped bring this book to market include the following:

Acquisitions and Editorial

Project Editor: Kelly Ewing

Senior Acquisitions Editor: Katie Feltman

Technical Editor: John Mueller

Editorial Manager: Jodi Jensen

Editorial Assistant: Amanda Graham

Sr. Editorial Assistant: Cherie Case

Cover Photo: © Javier Pierini / Jupiter Images

Cartoons: Rich Tennant (www.the5thwave.com)

Composition Services

Project Coordinator: Sheree Montgomery

Layout and Graphics: Carrie A. Cesavice, Corrie Niehaus, Lavonne Roberts

Proofreaders: ConText Editorial Services, Inc., Rebecca Denoncour

Indexer: Infodex Indexing Services, Inc.

Publishing and Editorial for Technology Dummies

Richard Swadley, Vice President and Executive Group Publisher

Andy Cummings, Vice President and Publisher

Mary Bednarek, Executive Acquisitions Director

Mary C. Corder, Editorial Director

Publishing for Consumer Dummies

Kathleen Nebenhaus, Vice President and Executive Publisher

Composition Services

Debbie Stailey, Director of Composition Services

Contents at a Glance

Introduction 1

Part I: Revving Up 7

Chapter 1: Getting Started 9
Chapter 2: Setting Up Your Computer 21
Chapter 3: Running Programs 47

Part II: Writing Your Own Java Programs 69

Chapter 4: Exploring the Parts of a Program 71
Chapter 5: Composing a Program 91
Chapter 6: Using the Building Blocks: Variables, Values, and Types 115
Chapter 7: Numbers and Types 129
Chapter 8: Numbers? Who Needs Numbers? 147

Part III: Controlling the Flow 169

Chapter 9: Forks in the Road 171
Chapter 10: Which Way Did He Go? 187
Chapter 11: How to Flick a Virtual Switch 209
Chapter 12: Around and Around It Goes 225
Chapter 13: Piles of Files: Dealing with Information Overload 245
Chapter 14: Creating Loops within Loops 265
Chapter 15: The Old Runaround 277

Part IV: Using Program Units 301

Chapter 16: Using Loops and Arrays 303
Chapter 17: Programming with Objects and Classes 321
Chapter 18: Using Methods and Variables from a Java Class 335
Chapter 19: Creating New Java Methods 357
Chapter 20: Oooey GUI Was a Worm 379

Part V: The Part of Tens 405

Chapter 21: Ten Sets of Web Links 407
Chapter 22: Ten Useful Classes in the Java API 413

Index 417

Table of Contents

Introduction 1

About This Book 1
How to Use This Book 2
Conventions Used in This Book 2
What You Don't Have to Read 3
Foolish Assumptions 3
How This Book Is Organized 4
Part I: Revving Up 4
Part II: Writing Your Own Java Programs 4
Part III: Controlling the Flow 5
Part IV: Using Program Units 5
Part V: The Part of Tens 5
Icons Used in This Book 5
Where to Go from Here 6

Part 1: Revving Up 7

Chapter 1: Getting Started 9

What's It All About? 9
Telling a computer what to do 10
Pick your poison 11
From Your Mind to the Computer's Processor 12
Translating your code 12
Running code 13
Code you can use 17
Your Java Programming Toolset 18
What's already on your hard drive? 20
Eclipse 20

Chapter 2: Setting Up Your Computer 21

If You Don't Like Reading Instructions 22
Getting This Book's Sample Programs 24
Setting Up Java 25
If you want to avoid installing Java 29
If you're juggling versions of Java on your computer 33
Setting Up the Eclipse Integrated Development Environment 35
Downloading Eclipse 35
Installing Eclipse 37
Running Eclipse for the first time 38
What's Next? 46

Chapter 3: Running Programs 47

Running a Canned Java Program 47
Typing and Running Your Own Code 52
Separating your programs from mine 52
Writing and running your program 53
What's All That Stuff in Eclipse's Window? 62
Understanding the big picture 62
Views, editors, and other stuff 63
What's inside a view or an editor? 65
Returning to the big picture 67

Part II: Writing Your Own Java Programs 69

Chapter 4: Exploring the Parts of a Program 71

Checking Out Java Code for the First Time 71
Behold! A program! 72
What the program's lines say 73
The Elements in a Java Program 73
Keywords 74
Identifiers that you or I can define 76
Identifiers with agreed-upon meanings 77
Literals 78
Punctuation 79
Comments 80
Understanding a Simple Java Program 82
What is a method? 82
The main method in a program 85
How you finally tell the computer to do something 85
The Java class 88

Chapter 5: Composing a Program 91

Computers Are Stupid 92
A Program to Echo Keyboard Input 92
Typing and running a program 94
How the EchoLine program works 96
Getting numbers, words, and other things 98
Type two lines of code and don't look back 100
Expecting the Unexpected 100
Diagnosing a problem 102
What problem? I don't see a problem 112

Chapter 6: Using the Building Blocks: Variables, Values, and Types 115

Using Variables 115
Using a variable 116
Understanding assignment statements 118
To wrap or not to wrap? 118
What Do All Those Zeros and Ones Mean? 119
Types and declarations 120
What's the point? 121
Reading Decimal Numbers from the Keyboard 122
Though these be methods, yet there is madness in 't 122
Methods and assignments 124
Variations on a Theme 124
Moving variables from place to place 125
Combining variable declarations 126

Chapter 7: Numbers and Types 129

Using Whole Numbers 129
Reading whole numbers from the keyboard 131
What you read is what you get 132
Creating New Values by Applying Operators 133
Finding a remainder 134
The increment and decrement operators 138
Assignment operators 143
Size Matters 144

Chapter 8: Numbers? Who Needs Numbers? 147

Characters 148
I digress 149
One character only, please 151
Variables and recycling 151
When not to reuse a variable 153
Reading characters 156
The boolean Type 157
Expressions and conditions 159
Comparing numbers; comparing characters 159
The Remaining Primitive Types 166

Part III: Controlling the Flow 169

Chapter 9: Forks in the Road 171

Decisions, Decisions! 171
Making Decisions (Java if Statements) 173
Looking carefully at if statements 173
A complete program 177
Indenting if statements in your code 179
Variations on the Theme 181
. . . Or else what? 181
Packing more stuff into an if statement 183
Some handy import declarations 186

Chapter 10: Which Way Did He Go? 187

Forming Bigger and Better Conditions 187
Combining conditions: An example 189
When to initialize? 191
More and more conditions 193
Using boolean variables 194
Mixing different logical operators together 196
Using parentheses 197
Building a Nest 199
Nested if statements 200
Cascading if statements 202
Enumerating the Possibilities 205
Creating an enum type 205
Using an enum type 206

Chapter 11: How to Flick a Virtual Switch 209

Meet the switch Statement 209
The cases in a switch statement 212
The default in a switch statement 213
Picky details about the switch statement 213
To break or not to break 217
Using Fall-Through to Your Advantage 219
Using a Conditional Operator 221

Chapter 12: Around and Around It Goes 225

Repeating Instructions Over and Over Again (Java while Statements) 226
Following the action in a loop 228
No early bailout 229

Thinking about Loops (What Statements Go Where) 230
Finding some pieces 231
Assembling the pieces 233
Getting values for variables 234
From infinity to affinity 235
Thinking about Loops (Priming) 238
Working on the problem 240
Fixing the problem 243

Chapter 13: Piles of Files: Dealing with Information Overload 245

Running a Disk-Oriented Program 246
A sample program 248
Creating code that messes with your hard drive 250
Running the sample program 253
Troubleshooting problems with disk files 255
Writing a Disk-Oriented Program 257
Reading from a file 258
Writing to a file 259
Writing, Rewriting, and Re-rewriting 262

Chapter 14: Creating Loops within Loops 265

Paying Your Old Code a Little Visit 266
Reworking some existing code 267
Running your code 268
Creating Useful Code 268
Checking for the end of a file 269
How it feels to be a computer 271
Why the computer accidentally pushes past the end of the file 273
Solving the problem 273

Chapter 15: The Old Runaround 277

Repeating Statements a Certain Number Times (Java for Statements) 278
The anatomy of a for statement 280
Initializing a for loop 281
Using Nested for Loops 284
Repeating Until You Get What You Need (Java do Statements) 286
Getting a trustworthy response 287
Deleting files 289
Using Java's do statement 291
A closer look at the do statement 291
Repeating with Predetermined Values (Java's Enhanced for Statement) 293
Creating an enhanced for loop 293
Nesting the enhanced for loops 295

Part IV: Using Program Units 301

Chapter 16: Using Loops and Arrays 303

Some Loops in Action 303
Deciding on a loop's limit at runtime 305
Using all kinds of conditions in a for loop 307
Reader, Meet Arrays; Arrays,Meet the Reader 309
Storing values in an array 313
Creating a report 314
Working with Arrays 316
Looping in Style 319

Chapter 17: Programming with Objects and Classes 321

Creating a Class 322
Reference types and Java classes 323
Using a newly defined class 323
Running code that straddles two separate files 325
Why bother? 325
From Classes Come Objects 326
Understanding (or ignoring) the subtleties 328
Making reference to an object's parts 329
Creating several objects 329
Another Way to Think about Classes 332
Classes, objects, and tables 332
Some questions and answers 334

Chapter 18: Using Methods and Variables from a Java Class 335

The String Class 335
A simple example 336
Putting String variables to good use 337
Reading and writing strings 338
Using an Object's Methods 339
Comparing strings 342
The truth about classes and methods 343
Calling an object's methods 345
Combining and using data 345
Static Methods 345
Calling static and non-static methods 346
Turning strings into numbers 347
Turning numbers into strings 349
How the NumberFormat works 350
Understanding the Big Picture 351
Packages and import declarations 352
Shedding light on the static darkness 353
Barry makes good on an age-old promise 354

Chapter 19: Creating New Java Methods 357

Defining a Method within a Class 357
Making a method.....358
Examining the method's header 359
Examining the method's body.....360
Calling the method.....360
The flow of control 362
Using punctuation.....363
The versatile plus sign 364
Let the Objects Do the Work.....366
Passing Values to Methods 368
Handing off a value 370
Working with a method header.....372
How the method uses the object's values.....372
Getting a Value from a Method 373
An example 374
How return types and return values work.....376
Working with the method header (again) 377

Chapter 20: Oooey GUI Was a Worm 379

The Java Swing Classes.....379
Showing an image on the screen 380
Just another class 382
Using Eclipse's WindowBuilder 384
Installing WindowBuilder.....385
Creating a GUI class.....387
Running your bare-bones GUI class 388
Show me the code.....389
Some details about the code 390
Adding Stuff to Your Frame.....395
Taking Action 401

Part V: The Part of Tens..... 405

Chapter 21: Ten Sets of Web Links 407

The Horse's Mouth.....407
Finding News, Reviews, and Sample Code 407
Improving Your Code with Tutorials 408
Finding Help on Newsgroups 408
Reading Documentation with Commentary.....408
Listen!.....409
Opinions and Advocacy.....409
Looking for Java Jobs.....409
Finding Out More about Other Programming Languages 410
Everyone's Favorite Sites 410

Chapter 22: Ten Useful Classes in the Java API 413

Applet 413
ArrayList 414
File 414
Integer 414
Math 415
NumberFormat 415
Scanner 415
String 416
StringTokenizer 416
System 416

Index 417

Introduction

What's your story?

- Are you a working stiff, interested in knowing more about the way your company's computers work?
- Are you a student who needs some extra reading in order to survive a beginning computer course?
- Are you a typical computer user — you've done lots of word processing, and you want to do something more interesting with your computer?
- Are you a job seeker with an interest in entering the fast-paced, glamorous, high-profile world of computer programming (or, at least, the decent-paying world of computer programming)?

Well, if you want to write computer programs, this book is for you. This book avoids the snobby "of-course-you-already-know" assumptions and describes computer programming from scratch.

About This Book

The book uses Java — a powerful, general-purpose computer programming language. But Java's subtleties and eccentricities aren't the book's main focus. Instead, this book emphasizes a process — the process of creating instructions for a computer to follow. Many highfalutin' books describe the mechanics of this process — the rules, the conventions, and the formalisms. But those other books aren't written for real people. Those books don't take you from where you are to where you want to be.

In this book, I assume very little about your experience with computers. As you read each section, you get to see inside my head. You see the problems that I face, the things that I think, and the solutions that I find. Some problems are the kind that I remember facing when I was a novice; other problems are the kind that I face as an expert. I help you understand, I help you visualize, and I help you create solutions on your own. I even get to tell a few funny stories.

How to Use This Book

I wish I could say, "Open to a random page of this book and start writing Java code. Just fill in the blanks and don't look back." In a sense, this is true. You can't break anything by writing Java code, so you're always free to experiment.

But I have to be honest. If you don't understand the bigger picture, writing a program is difficult. That's true with any computer programming language — not just Java. If you're typing code without knowing what it's about, and the code doesn't do exactly what you want it to do, then you're just plain stuck.

So in this book, I divide programming into manageable chunks. Each chunk is (more or less) a chapter. You can jump in anywhere you want — Chapter 5, Chapter 10, or wherever. You can even start by poking around in the middle of a chapter. I've tried to make the examples interesting without making one chapter depend on another. When I use an important idea from another chapter, I include a note to help you find your way around.

In general, my advice is as follows:

- If you already know something, don't bother reading about it.
- If you're curious, don't be afraid to skip ahead. You can always sneak a peek at an earlier chapter if you really need to do so.

Conventions Used in This Book

Almost every technical book starts with a little typeface legend, and *Beginning Programming with Java For Dummies,* 3rd Edition is no exception. What follows is a brief explanation of the typefaces used in this book:

- New terms are set in *italics.*
- When I want you to type something short or perform a step, I use **bold.**
- You'll also see this `computerese` font. I use the computerese font for Java code, filenames, web page addresses (URLs), onscreen messages, and other such things. Also, if something you need to type is really long, it appears in computerese font on its own line (or lines).
- You need to change certain things when you type them on your own computer keyboard. For example, I may ask you to type

  ```
  class Anyname
  ```

  which means you should type **class** and then some name that you make up on your own. Words that you need to replace with your own words are set in *`italicized computerese`*.

What You Don't Have to Read

Pick the first chapter or section that has material you don't already know and start reading there. Of course, you may hate making decisions as much as I do. If so, here are some guidelines you can follow:

- If you already know what computer programming is all about, then skip the first half of Chapter 1. Believe me, I won't mind.
- If you're required to use a development environment other than Eclipse, then you can skip Chapter 2. This applies if you plan to use NetBeans, IntelliJ IDEA, or a number of other development environments.

  Most of this book's examples require Java 5.0 or later, and some of the examples require Java 7 or later. So make sure that your system uses Java 5.0 or later. If you're not sure about your computer's Java version or if you have leeway in choosing a development environment, your safest move is to read Chapter 3.
- If you've already done a little computer programming, be prepared to skim Chapters 6 through 8. Dive fully into Chapter 9 and see whether it feels comfortable. (If so, then read on. If not, re-skim Chapters 6, 7, and 8.)
- If you feel comfortable writing programs in a language other than Java, then this book isn't for you. Keep this book as a memento and buy my *Java For Dummies,* 5th Edition, also published by John Wiley & Sons, Inc.

If you want to skip the sidebars and the Technical Stuff icons, then please do. In fact, if you want to skip anything at all, feel free.

Foolish Assumptions

In this book, I make a few assumptions about you, the reader. If one of these assumptions is incorrect, then you're probably okay. If all these assumptions are incorrect . . . well, buy the book anyway.

- **I assume that you have access to a computer.** Here's good news. You can run the code in this book on almost any computer. The only computers you can't use to run this code are ancient things that are more than 8 years old (give or take a few years). You can run the latest version of Java on Windows, Macintosh, and Linux computers.
- **I assume that you can navigate through your computer's common menus and dialog boxes.** You don't have to be a Windows, Linux, or Macintosh power user, but you should be able to start a program, find a file, put a file into a certain directory . . . that sort of thing. Most of the time, when you practice the stuff in this book, you're typing code on your keyboard, not pointing and clicking your mouse.

On those rare occasions when you need to drag and drop, cut and paste, or plug and play, I guide you carefully through the steps. But your computer may be configured in any of several billion ways, and my instructions may not quite fit your special situation. So when you reach one of these platform-specific tasks, try following the steps in this book. If the steps don't quite fit, send me an e-mail message, or consult a book with instructions tailored to your system.

- **I assume that you can think logically.** That's all there is to computer programming — thinking logically. If you can think logically, you've got it made. If you don't believe that you can think logically, read on. You may be pleasantly surprised.
- **I assume that you know little or nothing about computer programming.** This isn't one of those "all things to all people" books. I don't please the novice while I tease the expert. I aim this book specifically toward the novice — the person who has never programmed a computer or has never felt comfortable programming a computer. If you're one of these people, you're reading the right book.

How This Book Is Organized

This book is divided into subsections, which are grouped into sections, which come together to make chapters, which are lumped finally into five parts. (When you write a book, you get to know your book's structure pretty well. After months of writing, you find yourself dreaming in sections and chapters when you go to bed at night.) The parts of the book are listed here.

Part I: Revving Up

The chapters in Part I prepare you for the overall programming experience. In these chapters, you find out what programming is all about and get your computer ready for writing and testing programs.

Part II: Writing Your Own Java Programs

This part covers the basic building blocks — the elements in any Java program and in any program written using a Java-like language. In this part, you discover how to represent data and how to get new values from existing values. The program examples are short, but cute.

Part III: Controlling the Flow

Part III has some of my favorite chapters. In these chapters, you make the computer navigate from one part of your program to another. Think of your program as a big mansion, with the computer moving from room to room. Sometimes the computer chooses between two or more hallways, and sometimes the computer revisits rooms. As a programmer, your job is to plan the computer's rounds through the mansion. It's great fun.

Part IV: Using Program Units

Have you ever solved a big problem by breaking it into smaller, more manageable pieces? That's exactly what you do in Part IV of this book. You discover the best ways to break programming problems into pieces and to create solutions for the newly found pieces. You also find out how to use other peoples' solutions. It feels like stealing, but it's not.

This part also contains a chapter about programming with windows, buttons, and other graphical items. If your mouse feels ignored by the examples in this book, read Chapter 20.

Part V: The Part of Tens

The Part of Tens is a little beginning programmer's candy store. In the Part of Tens, you can find lists — lists of tips, resources, and all kinds of interesting goodies.

I added an appendix on this book's website to help you feel comfortable with Java's documentation. I can't write programs without my Java programming documentation. In fact, no Java programmer can write programs without those all-important docs. These docs are in web page format, so they're easy to find and easy to navigate. But if you're not used to all the terminology, the documentation can be overwhelming.

Icons Used in This Book

If you could watch me write this book, you'd see me sitting at my computer, talking to myself. I say each sentence several times in my head. When I have an extra thought, a side comment, something that doesn't belong in the regular stream, I twist my head a little bit. That way, whoever's listening to me (usually nobody) knows that I'm off on a momentary tangent.

Of course, in print, you can't see me twisting my head. I need some other way of setting a side thought in a corner by itself. I do it with icons. When you see a Tip icon or a Remember icon, you know that I'm taking a quick detour.

Here's a list of icons that I use in this book:

A tip is an extra piece of information — something helpful that the other books may forget to tell you.

Everyone makes mistakes. Heaven knows that I've made a few in my time. Anyway, when I think of a mistake that people are especially prone to make, I write about the mistake in a Warning icon.

Sometimes I want to hire a skywriting airplane crew. "Barry," says the white smoky cloud, "if you want to compare two numbers, use the double equal sign. Please don't forget to do this." Because I can't afford skywriting, I have to settle for something more modest. I create a Remember icon.

Occasionally, I run across a technical tidbit. The tidbit may help you understand what the people behind the scenes (the people who developed Java) were thinking. You don't have to read it, but you may find it useful. You may also find the tidbit helpful if you plan to read other (more geeky) books about Java.

This icon calls attention to useful material that you can find online. (You don't have to wait long to see one of these icons. I use one at the end of this introduction!)

Where to Go from Here

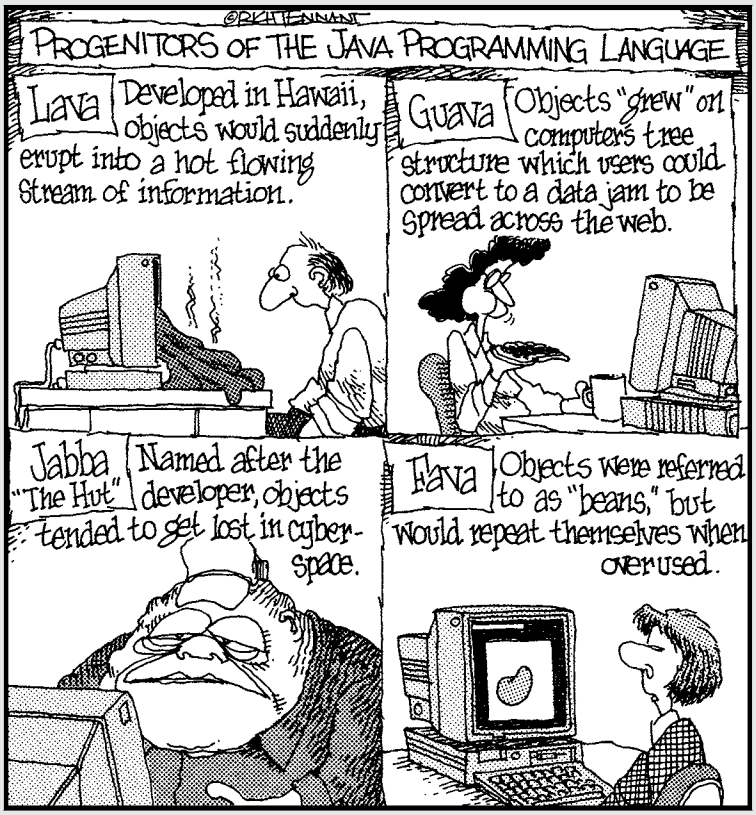
If you've gotten this far, then you're ready to start reading about computer programming. Think of me (the author) as your guide, your host, your personal assistant. I do everything I can to keep things interesting and, most importantly, help you understand.

If you like what you read, send me a note. My e-mail address, which I created just for comments and questions about this book, is `BeginProg@allmycode.com`. And don't forget — to get the latest information, visit one of this book's support websites. Mine is at `http://allmycode.com/BeginProg3`, or you can visit `www.dummies.com/go/beginningprogrammingwithjavafd`.

Occasionally, we have updates to our technology books. If this book does have technical updates, they will be posted at `www.dummies.com/go/beginningprogrammingwithjavafdupdates` and at `http://allmycode.com/BeginProg3`.

Part I
Revving Up

The 5th Wave By Rich Tennant


©RICH TENNANT
PROGENITORS OF THE JAVA PROGRAMMING LANGUAGE
Lava
Developed in Hawaii, objects would suddenly erupt into a hot flowing stream of information.
Guava
Objects "grew" on computers tree structure which users could convert to a data jam to be spread across the web.
Jabba "The Hut"
Named after the developer, objects tended to get lost in cyber-space.
Flava
Objects were referred to as "beans," but would repeat themselves when over used.

In this part . . .

You have to eat before you can cook. You have to wear before you can sew. You have to ride before you can drive. And you have to run computer programs before you can write computer programs.

In this part of the book, you run computer programs.

Chapter 1

Getting Started

In This Chapter

- Realizing what computer programming is all about
- Understanding the software that enables you to write programs
- Revving up to use an integrated development environment

Computer programming? What's that? Is it technical? Does it hurt? Is it politically correct? Does Bill Gates control it? Why would anyone want to do it? And what about me? Can I learn to do it?

What's It All About?

You've probably used a computer to do word processing. Type a letter, print it, and then send the printout to someone you love. If you have easy access to a computer, then you've probably surfed the web. Visit a page, click a link, and see another page. It's easy, right?

Well, it's easy only because someone told the computer exactly what to do. If you take a computer right from the factory and give no instructions to this computer, the computer can't do word processing, it can't surf the web, and it can't do anything. All a computer can do is follow the instructions that people give to it.

Now imagine that you're using Microsoft Word to write the great American novel, and you come to the end of a line. (You're not at the end of a sentence; just the end of a line.) As you type the next word, the computer's cursor jumps automatically to the next line of type. What's going on here?

Well, someone wrote a *computer program* — a set of instructions telling the computer what to do. Another name for a program (or part of a program) is *code.* Listing 1-1 shows you what some of Microsoft Word's code may look like.

Listing 1-1: A Few Lines in a Computer Program

```
if (columnNumber > 60) {
    wrapToNextLine();
} else {
    continueSameLine();
}
```

If you translate Listing 1-1 into plain English, you get something like this:

```
If the column number is greater than 60,
    then go to the next line.
Otherwise (if the column number isn't greater than 60),
    then stay on the same line.
```

Somebody has to write code of the kind shown in Listing 1-1. This code, along with millions of other lines of code, makes up the program called Microsoft Word.

And what about web surfing? You click a link that's supposed to take you directly to Yahoo.com. Behind the scenes, someone has written code of the following kind:

```
Go to <a href="http://www.yahoo.com">Yahoo</a>.
```

One way or another, someone has to write a program. That someone is called a *programmer*.

Telling a computer what to do

Everything you do with a computer involves gobs and gobs of code. Take a CD-ROM with a computer game on it. It's really a CD-ROM full of code. At some point, someone had to write the game program:

```
if (person.touches(goldenRing)) {
    person.getPoints(10);
}
```

Without a doubt, the people who write programs have valuable skills. These people have two important qualities:

- They know how to break big problems into smaller step-by-step procedures.
- They can express these steps in a very precise language.

A language for writing steps is called a *programming language,* and Java is just one of several thousand useful programming languages. The stuff in Listing 1-1 is written in the Java programming language.

Pick your poison

This book isn't about the differences among programming languages, but you should see code in some other languages so you understand the bigger picture. For example, there's another language, Visual Basic, whose code looks a bit different from code written in Java. An excerpt from a Visual Basic program may look like this:

```
If columnNumber > 60 Then
    Call wrapToNextLine
Else
    Call continueSameLine
End If
```

The Visual Basic code looks more like ordinary English than the Java code in Listing 1-1. But, if you think that Visual Basic is like English, then just look at some code written in COBOL:

```
IF COLUMN-NUMBER IS GREATER THAN 60 THEN
    PERFORM WRAP-TO-NEXT-LINE
ELSE
    PERFORM CONTINUE-SAME-LINE
END-IF.
```

At the other end of the spectrum, you find languages like Haskell. Here's a short Haskell program, along with the program's input and output:

```
median aList =
  [ x | x <- aList,
    length([y | y <- aList, y < x]) ==
    length([y | y <- aList, y > x])]
*Main> median [4,7,2,1,0,9,6]
[4]
```

Computer languages can be very different from one another, but, in some ways, they're all the same. When you get used to writing `IF COLUMN-NUMBER IS GREATER THAN 60`, then you can also become comfortable writing `if (columnNumber > 60)`. It's just a mental substitution of one set of symbols for another. Eventually, writing things like `if (columnNumber > 60)` becomes second nature.

From Your Mind to the Computer's Processor

When you create a new computer program, you go through a multistep process. The process involves three important tools:

- **Compiler:** A compiler translates your code into computer-friendly (human-unfriendly) instructions.
- **Virtual machine:** A virtual machine steps through the computer-friendly instructions.
- **Application programming interface:** An application programming interface contains useful prewritten code.

The next three sections describe each of the three tools.

Translating your code

You may have heard that computers deal with zeros and ones. That's certainly true, but what does it mean? Well, for starters, computer circuits don't deal directly with letters of the alphabet. When you see the word *Start* on your computer screen, the computer stores the word internally as `01010011 01110100 01100001 01110010 01110100`. That feeling you get of seeing a friendly looking five-letter word is your interpretation of the computer screen's pixels, and nothing more. Computers break everything down into very low-level, unfriendly sequences of zeros and ones and then put things back together so that humans can deal with the results.

So what happens when you write a computer program? Well, the program has to get translated into zeros and ones. The official name for the translation process is *compilation.* Without compilation, the computer can't run your program.

I compiled the code in Listing 1-1. Then I did some harmless hacking to help me see the resulting zeros and ones. What I saw was the mishmash in Figure 1-1.

The compiled mumbo jumbo in Figure 1-1 goes by many different names:

- Most Java programmers call it *bytecode.*
- I often call it a *.class file.* That's because, in Java, the bytecode gets stored in files named `SomethingOrOther.class`.
- To emphasize the difference, Java programmers call Listing 1-1 the *source code* and refer to the zeros and ones in Figure 1-1 as *object code.*

To visualize the relationship between source code and object code, see Figure 1-2. You can write source code and then get the computer to create object code from your source code. To create object code, the computer uses a special software tool called a *compiler*.

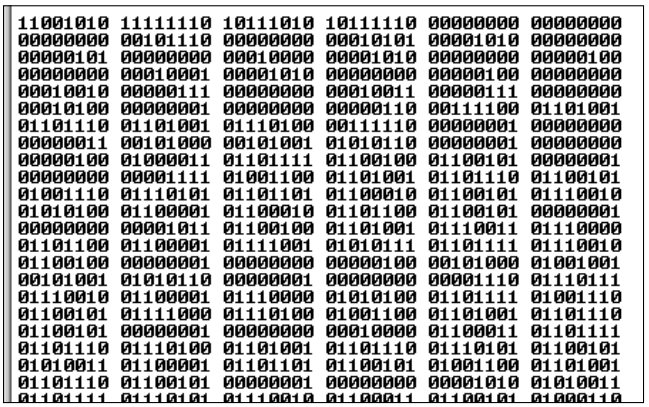


Figure 1-1: My computer understands these zeros and ones, but I don't.

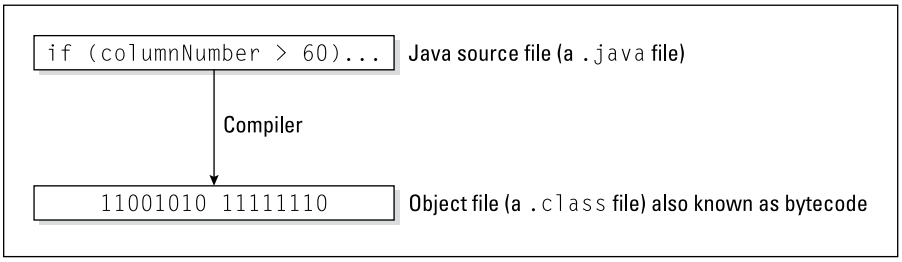

if (columnNumber > 60)...
Java source file (a .java file)
Compiler
11001010 11111110
Object file (a .class file) also known as bytecode

Figure 1-2: The computer compiles source code to create object code.

Your computer's hard drive may have a file named `javac` or `javac.exe`. This file contains that special software tool — the compiler. (Hey, how about that? The word `javac` stands for "Java compiler!") As a Java programmer, you often tell your computer to build some new object code. Your computer fulfills this wish by going behind the scenes and running the instructions in the `javac` file.

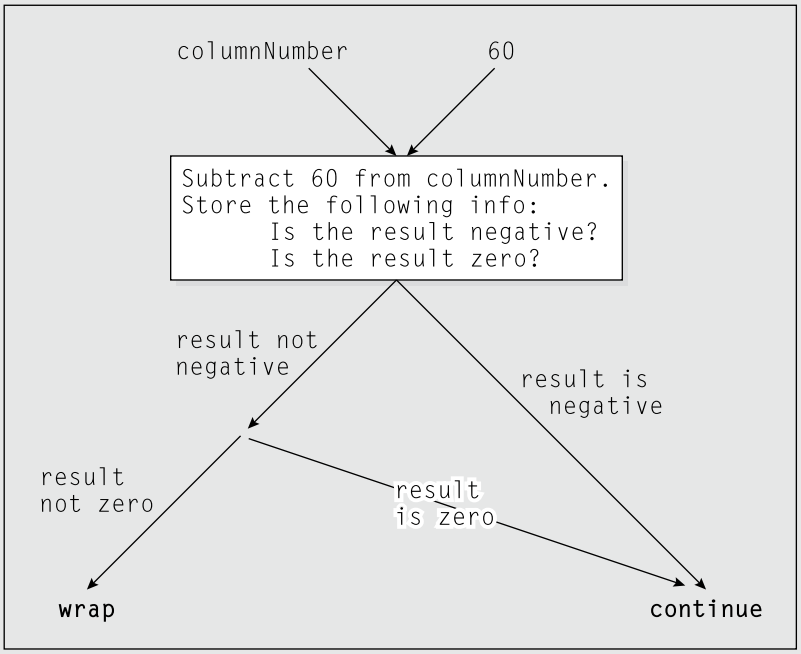
Running code

Several years ago, I spent a week in Copenhagen. I hung out with a friend who spoke both Danish and English fluently. As we chatted in the public park, I vaguely noticed some kids orbiting around us. I don't speak a word of Danish, so I assumed that the kids were talking about ordinary kid stuff.

What is bytecode, anyway?

Look at Listing 1-1 and at the listing's translation into bytecode in Figure 1-1. You may be tempted to think that a bytecode file is just a cryptogram — substituting zeros and ones for the letters in words like `if` and `else`. But it doesn't work that way at all. In fact, the most important part of a bytecode file is the encoding of a program's logic.

The zeros and ones in Figure 1-1 describe the flow of data from one part of your computer to another. I illustrate this flow in the following figure. But remember, this figure is just an illustration. Your computer doesn't look at this particular figure, or at anything like it. Instead, your computer reads a bunch of zeros and ones to decide what to do next.



Don't bother to absorb the details in my attempt at graphical representation in the figure. It's not worth your time. The thing you should glean from my mix of text, boxes, and arrows is that bytecode (the stuff in a `.class` file) contains a complete description of the operations that the computer is to perform. When you write a computer program, your source code describes an overall strategy — a big picture. The compiled bytecode turns the overall strategy into hundreds of tiny, step-by-step details. When the computer "runs your program," the computer examines this bytecode and carries out each of the little step-by-step details.

Then my friend told me that the kids weren't speaking Danish. "What language are they speaking?" I asked.

"They're talking gibberish," she said. "It's just nonsense syllables. They don't understand English, so they're imitating you."

Now to return to present-day matters. I look at the stuff in Figure 1-1, and I'm tempted to make fun of the way my computer talks. But then I'd be just like the kids in Copenhagen. What's meaningless to me can make perfect sense to my computer. When the zeros and ones in Figure 1-1 percolate through my computer's circuits, the computer "thinks" the thoughts shown in Figure 1-3.

Everyone knows that computers don't think, but a computer can carry out the instructions depicted in Figure 1-3. With many programming languages (languages like C++ and COBOL, for example), a computer does exactly what I'm describing. A computer gobbles up some object code and does whatever the object code says to do.

That's how it works in many programming languages, but that's not how it works in Java. With Java, the computer executes a different set of instructions. The computer executes instructions like the ones in Figure 1-4.

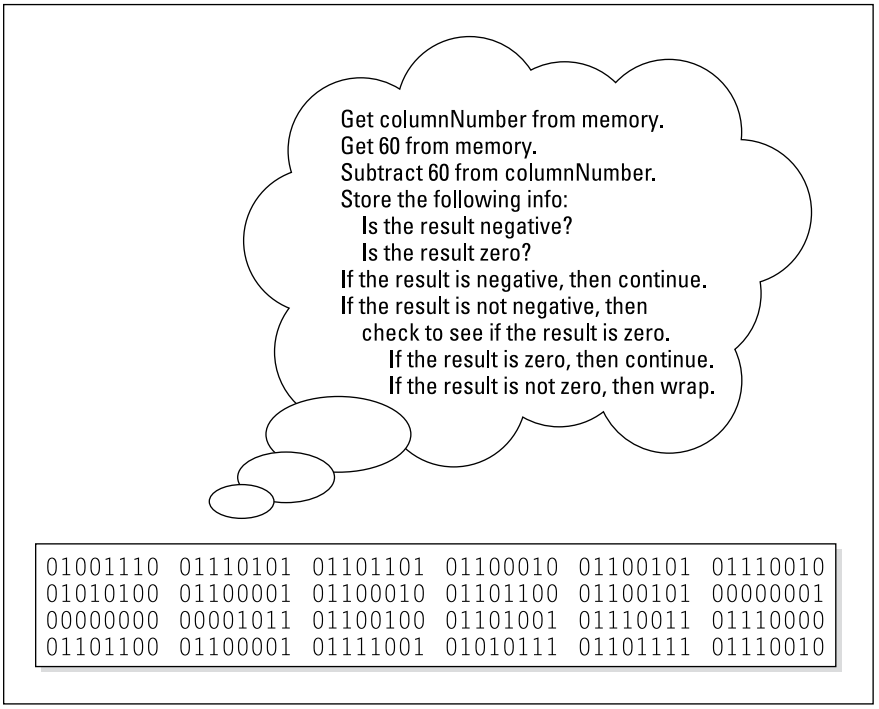

Get columnNumber from memory.
Get 60 from memory.
Subtract 60 from columnNumber.
Store the following info:
Is the result negative?
Is the result zero?
If the result is negative, then continue.
If the result is not negative, then
check to see if the result is zero.
If the result is zero, then continue.
If the result is not zero, then wrap.
01001110 01110101 01101101 01100010 01100101 01110010
01010100 01100001 01100010 01101100 01100101 00000001
00000000 00001011 01100100 01101001 01110011 01110000
01101100 01100001 01111001 01010111 01101111 01110010

Figure 1-3: What the computer gleans from a bytecode file.

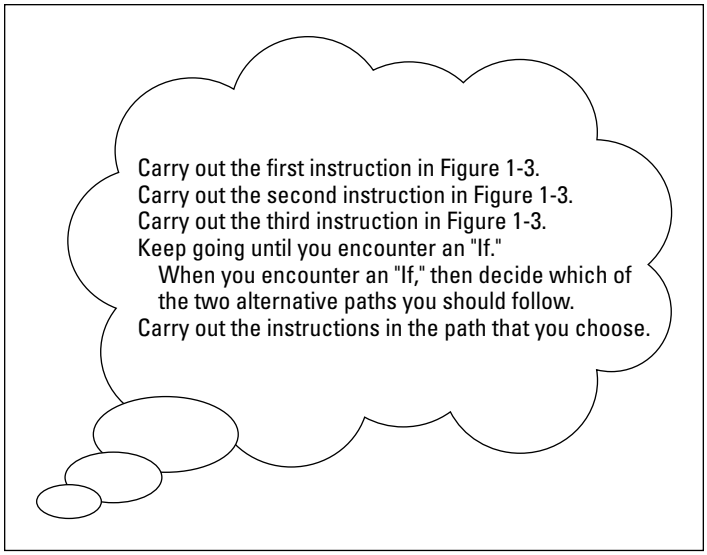

Carry out the first instruction in Figure 1-3.
Carry out the second instruction in Figure 1-3.
Carry out the third instruction in Figure 1-3.
Keep going until you encounter an "If."
When you encounter an "If," then decide which of the two alternative paths you should follow.
Carry out the instructions in the path that you choose.

Figure 1-4: How a computer runs a Java program.

The instructions in Figure 1-4 tell the computer how to follow other instructions. Instead of starting with `Get columnNumber from memory`, the computer's first instruction is, "Do what it says to do in the bytecode file." (Of course, in the bytecode file, the first instruction happens to be `Get columnNumber from memory`.)

There's a special piece of software that carries out the instructions in Figure 1-4. That special piece of software is called the *Java virtual machine* (JVM). The JVM walks your computer through the execution of some bytecode instructions. When you run a Java program, your computer is really running the JVM. That JVM examines your bytecode, zero by zero, one by one, and carries out the instructions described in the bytecode.

Many good metaphors can describe the JVM. Think of the JVM as a proxy, an errand boy, a go-between. One way or another, you have the situation shown in Figure 1-5. On the (a) side is the story you get with most programming languages — the computer runs some object code. On the (b) side is the story with Java — the computer runs the JVM, and the JVM follows the bytecode's instructions.

Your computer's hard drive may have a file named `java` or `java.exe`. This file contains the instructions illustrated previously in Figure 1-4 — the instructions in the JVM. As a Java programmer, you often tell your computer to run a Java program. Your computer fulfills this wish by going behind the scenes and running the instructions in the `java` file.

Write once, run anywhere

When Java first hit the tech scene in 1995, the language became popular almost immediately. This happened in part because of the JVM. The JVM is like a foreign language interpreter, turning Java bytecode into whatever native language a particular computer understands. So if you hand my Windows computer a Java bytecode file, then the computer's JVM interprets the file for the Windows environment. If you hand the same Java bytecode file to my colleague's Macintosh, then the Macintosh JVM interprets that same bytecode for the Mac environment.

Look again at Figure 1-5. Without a virtual machine, you need a different kind of object code for each operating system. But with the JVM, just one piece of bytecode works on Windows machines, Unix boxes, Macs, or whatever. This is called *portability,* and in the computer programming world, portability is a very precious commodity. Think about all the people using computers to browse the Internet. These people don't all run Microsoft Windows, but each person's computer can have its own bytecode interpreter — its own JVM.

The marketing folks at Oracle call it the *Write Once, Run Anywhere* model of computing. I call it a great way to create software.

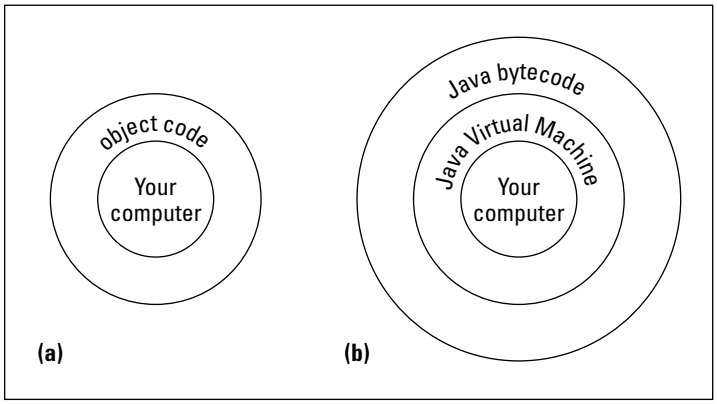


Figure 1-5: Two ways to run a computer program.

Code you can use

During the early 1980s, my cousin-in-law Chris worked for a computer software firm. The firm wrote code for word-processing machines. (At the time, if you wanted to compose documents without a typewriter, you bought a "computer" that did nothing but word processing.) Chris complained about being asked to write the same old code over and over again. "First, I write a search-and-replace program. Then I write a spell checker. Then I write another search-and-replace program. Then, a different kind of spell checker. And then, a better search-and-replace."

How did Chris manage to stay interested in his work? And how did Chris's employer manage to stay in business? Every few months, Chris had to reinvent the wheel. Toss out the old search-and-replace program and write a new program from scratch. That's inefficient. What's worse, it's boring.

For years, computer professionals were seeking the Holy Grail — a way to write software so that it's easy to reuse. Don't write and rewrite your search-and-replace code. Just break the task into tiny pieces. One piece searches for a single character, another piece looks for blank spaces, and a third piece substitutes one letter for another. When you have all the pieces, just assemble these pieces to form a search-and-replace program. Later on, when you think of a new feature for your word-processing software, you reassemble the pieces in a slightly different way. It's sensible, it's cost efficient, and it's much more fun.

The late 1980s saw several advances in software development, and by the early 1990s, many large programming projects were being written from prefab components. Java came along in 1995, so it was natural for the language's founders to create a library of reusable code. The library included about 250 programs, including code for dealing with disk files, code for creating windows, and code for passing information over the Internet. Since 1995, this library has grown to include more than 4,000 programs. This library is called the *Application Programming Interface* (*API*).

Every Java program, even the simplest one, calls on code in the Java API. This Java API is both useful and formidable. It's useful because of all the things you can do with the API's programs. It's formidable because the API is so extensive. No one memorizes all the features made available by the Java API. Programmers remember the features that they use often and look up the features that they need in a pinch. They look up these features in an online document called the *API Specification* (known affectionately to most Java programmers as the *API documentation*, or the *Javadocs*).

The API documentation describes the thousands of features in the Java API. As a Java programmer, you consult this API documentation on a daily basis. You can bookmark the documentation at the Oracle website and revisit the site whenever you need to look up something. But in the long run (and in the not-so-long run), you can save time by downloading your own copy of the API docs. (For details, see Chapter 2.)

Your Java Programming Toolset

To write Java programs, you need the tools described previously in this chapter:

- **You need a Java compiler.** (See the section "Translating your code.")
- **You need a JVM.** (See the section "Running code.")

- **You need the Java API.** (See the section "Code you can use.")
- **You need access to the Java API documentation.** (Again, see the "Code you can use" section.)

You also need some less exotic tools:

- **You need an editor to compose your Java programs.** Listing 1-1 contains part of a computer program. When you come right down to it, a computer program is a big bunch of text. So to write a computer program, you need an *editor* — a tool for creating text documents.

  An editor is a lot like Microsoft Word, or like any other word-processing program. The big difference is that an editor adds no formatting to your text — no bold, italic, or distinctions among fonts. Computer programs have no formatting whatsoever. They have nothing except plain old letters, numbers, and other familiar keyboard characters.

  When you edit a program, you may see bold text, italic text, and text in several colors. But your program contains none of this formatting. If you see stuff that looks like formatting, it's because the editor that you're using does *syntax highlighting*. With syntax highlighting, an editor makes the text appear to be formatted in order to help you understand the structure of your program. Believe me, syntax highlighting is very helpful.

- **You need a way to issue commands.** You need a way to say things like "compile this program" and "run the JVM." Every computer provides ways of issuing commands. (You can double-click icons or type verbose commands in a Run dialog box.) But when you use your computer's facilities, you jump from one window to another. You open one window to read Java documentation, another window to edit a Java program, and a third window to start up the Java compiler. The process can be very tedious.

In the best of all possible worlds, you do all your program editing, documentation reading, and command issuing through one nice interface. This interface is called an *integrated development environment* (*IDE*).

A typical IDE divides your screen's work area into several panes — one pane for editing programs, another pane for listing the names of programs, a third pane for issuing commands, and other panes to help you compose and test programs. You can arrange the panes for quick access. Better yet, if you change the information in one pane, the IDE automatically updates the information in all the other panes.

An IDE helps you move seamlessly from one part of the programming endeavor to another. With an IDE, you don't have to worry about the mechanics of editing, compiling, and running a JVM. Instead, you can worry about the logic of writing programs. (Wouldn't you know it? One way or another, you always have something to worry about!)

What's already on your hard drive?

You may already have some of the tools you need for creating Java programs. Here are some examples:

- **Some Windows installations already have a JVM.** Look for a file named `java.exe` in your `\windows\system32` directory.
- **Computers running versions of Mac OS X up to and including 10.6 (Snow Leopard) come with a Java compiler, a JVM, and a Java API.**
- **Computers running Mac OS X 10.7 (Lion) don't come with a JVM.** To install a JVM, visit `www.macupdate.com/app/mac/39490/java-for-os-x-lion`
- **Some IDEs come with their own Java tools.** For example, when you download the free Eclipse IDE you get a Java compiler and a copy of the Java API documentation. (For details, see Chapter 2.)

You may already have some Java tools but, on an older computer, your tools may be obsolete. Most of this book's examples run on all versions of Java. But some examples don't run on versions earlier than Java 5.0. Other examples run only on Java 6 or later, and some examples run only on Java 7 or later.

The safest bet is to download tools afresh from the Oracle website. To get detailed instructions on doing the download, see Chapter 2.

Eclipse

The programs in this book work with any IDE that can run Java. This includes IDEs such as NetBeans, IntelliJ IDEA, JDeveloper, JCreator, and others. You can even run the programs without an IDE. But to illustrate the examples in this book, I use the Eclipse IDE. I chose Eclipse over other IDEs for several reasons:

- Eclipse is free.
- Among all the Java IDEs, Eclipse is the one most commonly used by professional programmers.
- Eclipse has many bells and whistles, but you can ignore most of them and learn to repeat a few routine sequences of steps. After using Eclipse a few times, your brain automatically performs the routine steps. From then on, you can stop worrying about Eclipse and concentrate on Java programming.
- Eclipse is free. (It's worth mentioning twice.)

Chapter 2

Setting Up Your Computer

In This Chapter

- Installing Java
- Downloading and installing the Eclipse integrated development environment
- Checking your Eclipse configuration
- Getting the code in this book's examples

This book tells you how to write Java programs, and before you can write Java programs, you need some software. At the very least, you need the software that I describe in Chapter 1 — a Java compiler and a Java virtual machine (JVM, for short). You can also use a good integrated development environment (IDE) and some sample code to get you started.

You can get some of this software in a brightly colored box, but it's easier (and cheaper) to download the software from the web. In fact, all the software you need for writing Java programs is free. The software comes as three downloads — one from this book's website, another from Oracle, and a third from `eclipse.org`. Who needs another brightly colored box, anyway?

The Oracle and Eclipse websites that I describe in this chapter are always changing. The software that you download from these sites changes, too. A specific instruction such as "click the button in the upper-right corner" becomes obsolete (and even misleading) in no time at all. So in this chapter, I provide long list of steps, but I also describe the ideas behind the steps. Browse each of the suggested sites and look for ways to get the software that I describe. When a website offers you several options, check the instructions in this chapter for hints on choosing the best option. If your computer's Eclipse window doesn't look quite like the window in this chapter's figures, scan your computer's window for whatever options I describe. If, after all that, you can't find what you're looking for, check this book's website (`http://allmycode.com/BeginProg3`) or send an e-mail to me at `BeginProg@allmycode.com`.

If You Don't Like Reading Instructions . . .

I start this chapter with a very brief (but useful) overview of the steps required to get the software you need. If you're an old hand at installing software, and if your computer isn't quirky, these steps will probably serve you well. If not, you can read the more detailed instructions in the next several sections.

Here's how you get the software for creating Java programs:

1. **Visit** `http://allmycode.com/BeginProg3` **and download a file containing all the program examples in this book.**

2. **Visit** `www.oracle.com/technetwork/java/javase/downloads` **and get the latest available version of the JRE or the JDK.**

   If you plan to use Eclipse (as I describe throughout this book), either the JRE or the JDK is fine. Choose a version of the software that matches your operating system (Windows, Macintosh, or whatever). If you have trouble choosing between 32-bit software and 64-bit software, the 32-bit versions make safer choices.

   If you're in a hurry (and who isn't?), you may benefit from a quick visit to `http://java.com`. The `http://java.com` website offers a hassle-free, one-click Java installer. (Simply click a big Java Download button. You can't miss it.) The Java Download button doesn't work on all computers. But if it works for you, then with a wave of a virtual magic wand, you're finished with this step. You can bypass the complexities of the `oracle.com` website and move immediately to Step 3.

3. **Visit** `http://eclipse.org/downloads` **and get the Eclipse IDE.**

   Select the Eclipse IDE For Java Developers. The resulting download is a compressed archive file (for Windows, a `.zip` file; for other operating systems, including Macintosh OS X, a `.tar.gz` file).

4. **Extract the contents of the downloaded Eclipse archive.**

   The archive contains a folder named `eclipse`. Extract this `eclipse` folder to a handy place in your computer's hard drive. For example, on my Windows computer, I have a `C:\eclipse` folder. On my Mac, I have an `eclipse` folder inside my `Applications` folder.

   In Windows, the blank space in the name `Program Files` confuses some Java software. I don't think any of this book's software presents such a problem, but I can't guarantee it. So if you want, extract Eclipse to your `C:\Program Files` or `C:\Program Files (x86)` folder. But

make a mental note about your choice (in case you run into any trouble later).

5. **Launch Eclipse and click the Welcome screen's Workbench icon.**

   The Welcome screen's icons have no text labels. But when you hover over an icon, a tooltip appears. Select the icon whose tooltip has the title Workbench.

6. **In Eclipse, import the code that you downloaded in Step 1.**

For details about any of this stuff, see the next several sections.

Those pesky filename extensions

The filenames displayed in My Computer or in a Finder window can be misleading. You may browse one of your directories and see the name `Mortgage`. The file's real name might be `Mortgage.java`, `Mortgage.class`, `Mortgage.`*`somethingElse`*, or plain old `Mortgage`. Filename endings like `.zip`, `.java` and `.class` are called *filename extensions*.

The ugly truth is that, by default, Windows and Macs hide many filename extensions. This awful feature tends to confuse programmers. So, if you don't want to be confused, change your computer's system-wide settings. Here's how you do it:

- **In Windows XP:** Choose Start➪Control Panel➪Appearance and Themes➪Folder Options. Then skip to the **all versions of Windows** bullet.
- **In Windows 7:** Choose Start➪Control Panel➪Appearance and Personalization➪Folder Options. Then skip to the **all versions of Windows** bullet.
- **In Windows 8:** In the Start screen, select Control Panel➪More Settings➪Appearance and Personalization➪Folder Options. Then proceed to the **all versions of Windows** bullet.
- **In all versions of Windows (XP and newer):** Follow the instructions in one of the preceding bullets. Then, in the Folder Options dialog box, click the View tab. Look for the Hide File Extensions For Known File Types option. Make sure that this check box is *not* selected.
- **In Mac OS X:** In the Finder application's menu, select Preferences. In the resulting dialog box, select the Advanced tab and look for the Show All File Extensions option. Make sure that this check box *is* selected.
- **In Linux:** Linux distributions tend not to hide filename extensions. So if you use Linux, you probably don't have to worry about this. But I haven't checked all Linux distributions. So if your files are named `Mortgage` instead of `Mortgage.java` or `Mortgage.class`, check the documentation specific to your Linux distribution.

Getting This Book's Sample Programs

To get copies of this book's sample programs, visit `http://allmycode.com/BeginProg3` and click the link to download the programs in this book. Save the download file (`BeginProgJavaDummies3.zip`) to your computer's hard drive.

In some cases, you click a download link, but your web browser doesn't offer you the option to save a file. If this happens to you, right-click the link (or control-click on a Mac). In the resulting context menu, select Save Target As, Save Link As, Download Linked File As, or a similarly labeled menu item.

Most web browsers save files to a `Downloads` directory on your computer's hard drive. But your browser may be configured a bit differently. One way or another, make note of the folder containing the downloaded `BeginProgJavaDummies3.zip` file.

Compressed archive files

When you visit `http://allmycode.com/BeginProg3` and you download this book's Java examples, you download a file named `BeginProgJavaDummies3.zip`. A `.zip` file is a single file that encodes a bunch of smaller files and folders. So, for example, my `BeginProgJavaDummies3.zip` file encodes folders named `06-01`, `06-02`, and so on. The `06-02` folder contains some subfolders, which in turn contain files. (The folder named `06-02` contains the code in Listing 6-2 — the second listing in Chapter 6.)

A `.zip` file is an example of a *compressed archive* file. Some other examples of compressed archives include `.tar.gz` files, `.rar` files, and `.cab` files. *Uncompressing* a file means extracting the original files stored inside the big archive file. (For a `.zip` file, another word for "uncompressing" is "*unzipping*.") Uncompressing normally re-creates the folder structure encoded in the archive file. So after uncompressing my `BeginProgJavaDummies3.zip` file, your hard drive has folders named `06-01`, `06-02`, with subfolders named `src` and `bin`, which in turn contain files named `SnitSoft.java`, `SnitSoft.class`, and so on.

When you download `BeginProgJavaDummies3.zip`, your web browser may uncompress the file automatically for you. If not, you can see the `.zip` file's contents by double-clicking the file's icon. (In fact, you can copy the file's contents and perform some other file operations after double-clicking the file's icon.) One way or another, don't worry about uncompressing my `BeginProgJavaDummies3.zip` file. When you follow this chapter's instructions, you import the contents of my `BeginProgJavaDummies3.zip` file into the Eclipse IDE. And behind the scenes, Eclipse's import process uncompresses the `.zip` file.

Setting Up Java

You can get the latest, greatest versions of Java by visiting `www.oracle.com/technetwork/java/javase/downloads`. Look for the newest available version of the JRE or the JDK. Select a version that runs on your computer's operating system. Figure 2-1 shows me clicking a Download JRE button (*circa* November 2011) at the `http://oracle.com` website.

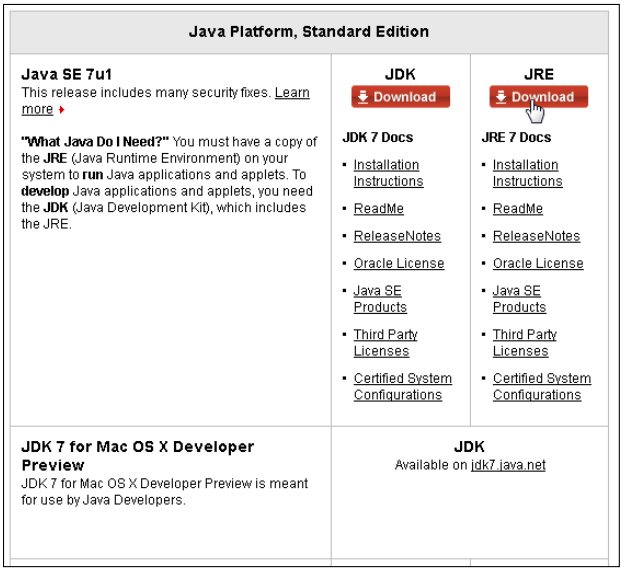

Java Platform, Standard Edition
Java SE 7u1
This release includes many security fixes. Learn more
"What Java Do I Need?" You must have a copy of the JRE (Java Runtime Environment) on your system to run Java applications and applets. To develop Java applications and applets, you need the JDK (Java Development Kit), which includes the JRE.
JDK
Download
JDK 7 Docs
Installation Instructions
ReadMe
ReleaseNotes
Oracle License
Java SE Products
Third Party Licenses
Certified System Configurations
JRE
Download
JRE 7 Docs
Installation Instructions
ReadMe
ReleaseNotes
Oracle License
Java SE Products
Third Party Licenses
Certified System Configurations
JDK 7 for Mac OS X Developer Preview
JDK 7 for Mac OS X Developer Preview is meant for use by Java Developers.
JDK
Available on jdk7.java.net

Figure 2-1: Getting the Java JRE.

If you can't identify the most appropriate Java version or if you want to know what the acronyms *JRE* and *JDK* stand for, see the sidebar entitled "Eenie, meenie, miney mo."

After you accept a license agreement and click a link to a Java installation file, your computer does one of two things:

- **Downloads and installs Java on your system.**
- **Downloads the Java installation file and saves the file on your computer's hard drive.**

Eenie, meenie, miney mo

The Java Standard Edition download page (`www.oracle.com/technetwork/java/javase/downloads`) has many options. If you're not familiar with these options, the page can be intimidating. Here are some of the choices on the page:

- **Word length: 32-bit or 64-bit**

  You may have to choose between links labeled for 32-bit systems and links labeled for 64-bit systems. If you don't know which to choose, then start by trying the 32-bit version. (For more information about 32-bit systems and 64-bit systems, see the "How many bits does your computer have?" sidebar.)

- **Java version number**

  The Java download page may have older and newer Java versions for you to choose from. You may see links to Java SE 6, Java SE 7, Java SE 7u4, and many others. (Numbering such as *7u4* stands for the fourth update to Java 7.) If you're not sure which version number you want, choosing the highest version number is probably safe. Most of this book's examples run on a computer with Java 5 installed. A few examples run only on Java 7 or higher.

  The numbering of Java's versions is really confusing. First comes Java 1.0, then Java 1.1, and then Java 2 Standard Edition 1.2 (J2SE 1.2). Yes, the "Java 2" numbering overlaps partially with the "1.x" numbering. Next come versions 1.3 and 1.4. After version 1.4.1 comes version 1.4.2 (with intermediate stops at versions like 1.4.1_02). After 1.4.2_06, the next version is version 1.5, which is also known as version 5.0. (That's no misprint. Version 5.0 comes immediately after the 1.4 versions.)

  The formal name for version 1.5 is "Java 2 Platform, Standard Edition 5.0." And to make matters even worse, the next big release is "Java Platform, Standard Edition 6" with the "2" removed from "Java 2" and the ".0" missing from "6.0." That's what happens when a company lets marketing people call the shots.

  Mercifully, from Java 6 onward, the version numbers settle into a predictable pattern. After Java 6 comes "Java Platform, Standard Edition 7" with updates such as "7u4" (meaning "Java 7, update 4").

- **JDK versus JRE**

  The download page offers you a choice between the JDK (Java Development Kit) and the JRE (Java Runtime Environment). The JDK download contains more stuff than the JRE download. The JRE includes a Java virtual machine and the Application Programming Interface (see Chapter 1). The JDK includes everything in the JRE and, in addition, the JDK includes a Java compiler.

  You can download the JDK, but instead I recommend downloading the JRE. The JRE download is smaller, and the Eclipse IDE contains its own Java compiler. So when you get the JRE and Eclipse (per this chapter's instructions), you get everything you need to write your own Java programs.

  By the way, another name for the JDK is the *Java SDK* — the *Java Software Development Kit.* Some people still use the SDK acronym, even though the folks at Oracle don't use it anymore. (Actually, the original name was the JDK. Later, Sun Microsystems changed it to the SDK. A few years after that, the captains of Java changed back to the name JDK. This constant naming and renaming drives me crazy as an author.)

- **Java SE versus Java EE versus Java ME**

  While you wander around, you may notice links labeled Java EE or Java ME. If you know what these are, and you know you need them, then by all means, download these goodies. But if you're not sure, then bypass both the Java EE and the Java ME links. Instead, follow links to the Java SE (Java Standard Edition).

  The abbreviation Java EE stands for Java Enterprise Edition and Java ME stands for Java Micro Edition. The Enterprise Edition has software for large businesses, and the Micro Edition has software for handheld devices. (Google's Android software bares a passing resemblance to Java's Micro Edition, but in many ways, Android and Java ME are very different animals.)

  You don't need the Java EE or the Java ME to run any of the examples in this book.

- **Additional Java-related software**

  You can download Java alone, or you can download Java with Oracle's NetBeans IDE. You can download JavaFX — a high-powered platform for creating games and applications. You can probably even download Java with fries and a soft drink. You can download plenty of extra stuff, but in truth, all you need is the Java JRE.

- **Installation type**

  You may be prompted to choose between online installation and offline installation.

  With the offline installation, you begin by downloading a large setup file. The file takes up lots of space on your hard drive (between 10 MB and 150 MB, depending on what you choose to download). If you ever need to install the JDK again, you have the file on your own computer. Until you update your version of the JDK, you don't need to download the JDK again.

  Why would anyone want to install the same version of the JDK a second time? Typically, I have two reasons. Either I want to install the software on a second computer, or I mess something up and have to uninstall (and then reinstall) the software.

  With the online installation, you don't download a big setup file. Instead, you download a teeny little setup file. Then you download (and discard) pieces of the big 10 to 150MB file as you need them. Using online installation saves you many megabytes of hard drive space. But, if you want to install the same version of the JDK a second time, you have to redo the whole surf/click/download process.

If the installation begins on its own, then follow the instructions, answer "Yes" to any prompts, and (unless you have good reason to do otherwise) accept the defaults. If the installation doesn't begin on its own, start by double-clicking the downloaded installation file.

- **On Windows:** Accept the defaults offered by the installation wizard.
- **On a Mac:** A window urges you to drag the JDK file to a `JavaVirtualMachines` folder. Before you can drag anything to the `JavaVirtualMachines` folder, you may have to retype your computer's password.
- **With a Linux `.rpm` file:** Accept the defaults offered by the installation wizard.

- **With a Linux `.tar.gz` file:** A `.tar.gz` file is a compressed archive. Extract the archive's contents to a folder of your choice. (For more information about filenames and file types, see the sidebars entitled "Those pesky filename extensions" and "Compressed archive files" in this chapter.)

Most people have no difficulties visiting `http://oracle.com` and installing Java using the website's menus. But if your situation is more "interesting" than most, you may have to make some decisions and perform some extra steps. The next few sections describe some of these "interesting" scenarios.

How many bits does your computer have?

As you follow this chapter's instructions, you may be prompted to choose between two versions of a piece of software — the 32-bit version and the 64-bit version. What's the difference, and why should you care?

A *bit* is the smallest piece of information that you can store on a computer. Most people think of a bit as either a zero or a one, and that depiction of "bit" is quite useful. To represent almost any number, you pile several bits next to one another and do some fancy things with powers of two. The numbering system's details aren't show stoppers. The important thing to remember is that each piece of circuitry inside your computer stores the same number of bits. (Well, some circuits inside your computer are outliers with their own particular numbers of bits, but that's not a big deal.)

In an older computer, each piece of circuitry stores 32 bits. In a newer computer, each piece of circuitry stores 64 bits. This number of bits (either 32 or 64) is the computer's *word length*. In a newer computer, a word is 64 bits long.

"Great!" you say. "I bought my computer last week. It must be a 64-bit computer." Well, the story may not be that simple. In addition to your computer's circuitry having a word length, the operating system on your computer also has a word length. An operating system's instructions work with a particular number of bits. An operating system with 32-bit instructions can run on either a 32-bit computer or a 64-bit computer, but an operating system with 64-bit instructions can run only on a 64-bit computer. And to make things even more complicated, each program that you run (a web browser, a word processor, or one of your own Java programs) is either a 32-bit program or a 64-bit program. You may run a 32-bit web browser on a 64-bit operating system running on a 64-bit computer. Alternatively, you may run a 32-bit browser on a 32-bit operating system on a 64-bit computer. (See the figure that accompanies this sidebar.)

When a website makes you choose between 32-bit and 64-bit software versions, the main consideration is the word length of your operating system, not the word length of your computer's circuitry. You can run a 32-bit word processor on a 64-bit operating system, but you can't run a 64-bit word processor on a 32-bit operating system (no matter what word length your computer's circuitry has). Choosing 64-bit software has one big advantage — namely, that 64-bit software can access more than 3 gigabytes of a computer's fast random access memory. And in my experience, more memory means faster processing.

How does all this stuff about word lengths affect your Java and Eclipse downloads? Here's the story:

- If you run a 32-bit operating system, you run only 32-bit software.

- If you run a 64-bit operating system, you probably run some 32-bit software and some 64-bit software. Most 32-bit software runs fine on a 64-bit operating system.
- On a 64-bit operating system, you might have two versions of the same program. For example, on my Windows computer, I have two versions of Internet Explorer — a 32-bit version and a 64-bit version.

  Normally, Windows puts 32-bit programs in its `Program Files (x86)` directory and puts 64-bit programs in its `Program Files` directory.
- A chain of word lengths is as strong as its weakest link. For example, when I visit `java.com` and click the site's *Do I have Java?* link, the answer I get depends on the match between my computer's Java version and the web browser that I'm running.

  With only 64-bit Java installed on my computer, the *Do I have Java?* link in my 32-bit Firefox browser answers *No working Java was detected on your system*. But the same link in my 64-bit Internet Explorer answers *You have the recommended Java installed*.
- Here's the most important thing to remember about word lengths: When you follow this chapter's instructions, you get Java software and Eclipse software on your computer. Your Java software's word length must match your Eclipse software's word length. In other words, 32-bit Eclipse runs with 32-bit Java, and 64-bit Eclipse runs with 64-bit Java. I haven't tried all possible combinations, but when I try to run 32-bit Eclipse with 64-bit Java, I see a misleading "No Java virtual machine was found" error message.

If you want to avoid installing Java . . .

Chapter 1 describes the Java ecosystem with its compiler, its virtual machine, and its other parts. Your computer may already have some of these Java gizmos. If so, you can either live with what you already have or add the newest version of Java to whatever is already on your system. To find out what you already have and possibly avoid reinstalling Java, keep reading:

On Windows

Select Start➪Control Panel➪Programs➪Programs and Features and look for *Java* in the list of installed programs (see Figure 2-2). If you see Java 7 or higher, you're okay. If the only Java version numbers that you see are less than 7 (such as 1.4.2, 5.0, or 6), then your computer doesn't have a version that runs all of this book's programs.

If the version number is 5.0 or higher, you can run many (but not all) of the programs in this book.

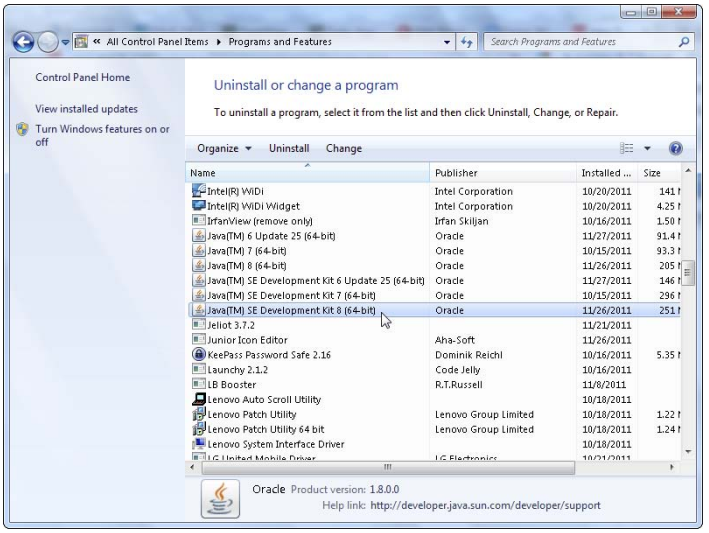


Figure 2-2: The Programs and Features dialog box on Windows 7.

On a Mac

Table 2-1 describes the correspondence between up-to-date Mac OS versions and Java versions.

Table 2-1 Mac OS X Versions and Java Versions

If You Have This Mac OS X Version . . .	*Then You Have This Version of Java . . .*	*And You Can Run This Java Version*
OS X 10.4 (Tiger)	Java 5.0	Java 5.0
OS X 10.5 (Leopard) PowerPC and/or 32-bit	Java 5.0	Java 5.0
OS X 10.5 (Leopard) Intel-based and 64-bit	Java 6	Java 6
OS X 10.6 (Snow Leopard) 32-bit	Java 6	Java 6
OS X 10.6 (Snow Leopard) 64-bit	Java 6	Java 7
OS X 10.7 (Lion)	Java 6	Java 7

Tiger, Leopard, and Snow Leopard have Java preinstalled. If you apply software updates when you're prompted to do so, you have whatever version of Java you find in Table 2-1. Java isn't preinstalled on Lion. Instead, the system automatically installs Java 6 the first time you launch an application that requires Java.

To find out which version of OS X you're running, do the following:

1. **Choose Apple➪About This Mac.**
2. **In the About This Mac dialog that appears, look for the word *Version*.**

   You see Version 10.6.4 (or something like that) in very light grey text.

If you don't regularly apply software updates, select Software Update in the Apple menu. In the resulting Software Update dialog, click the Show Details button. Then select items with the word *Java* in them (see Figure 2-3). If you suspect that an item relates to Java, but you're not sure, you can highlight the item to see the item's detailed description. In any case, you can select more items than you need, including items having nothing to do with Java. After making your selections, click the Install button.

Software Update
New software is available for your computer.
If you don't want to install now, choose Apple menu > Software Update when you're ready to install.
Install
Name
Version
Size
Safari
5.1.2
49.2 MB
Java for Mac OS X 10.6 Update 6
6.0
79.2 MB
Security Update 2011-006
1.0
142.5 MB
Java for Mac OS X 10.6 Update 6 delivers improved compatibility, security, and reliability by updating Java SE 6 to 1.6.0_29.
Please quit any web browsers and Java applications before installing this update.
See http://support.apple.com/kb/HT4884 for more details about this update.
See http://support.apple.com/kb/HT1222 for information about the security content of this update.
Note: Use of this software is subject to the original Software License Agreement(s) that accompanied the software being updated. A list of Apple SLAs may be found here: http://www.apple.com/legal/sla/.
Java for Mac OS X 10.6 Update 6: Downloaded.
You must restart your computer after the updates are installed.
Hide Details
Not Now
Install 2 Items

Figure 2-3: Updating Java and other important software.

Here and there on the web, I see postings describing ways to install Java 5.0 on OS X 10.3, and other ways to circumvent the restrictions in Table 2-1. But if you don't like to tinker, these workarounds aren't for you. (For every hardware or software requirement, someone tries to create a workaround, or *hack*. Anyway, apply hacks at your own risk.)

If you don't trust Table 2-1 (and frankly, you shouldn't trust everything you find in print), you can perform tests on your computer to discover the presence of Java and (if your Mac has Java) the Java version number. Here's one such test:

1. **In the Spotlight's search field, type** Java Preferences**.**
2. **When the Spotlight's Top Hit is Java Preferences, press Enter.**

   The Java Preferences window appears (see Figure 2-4).

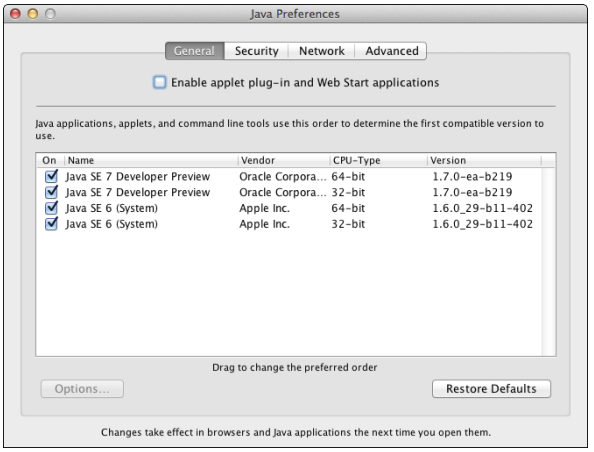


Figure 2-4: The Java Preferences application.

3. **The Java Preferences window lists all versions of Java that are installed on your computer.**

   In Figure 2-4, the computer has four versions of Java — the 32- and 64-bit versions of Java 6 and Java 7.

On Linux

To check your Java installation (or your lack of Java) on a Linux computer, do the following:

1. **Poke around among the desktop's menus for something named Terminal (also known as Konsole).**

   A Terminal window opens (usually with plain white text on a plain black background).

2. **In the Terminal window, type the following text and then press Enter:** java -version**.**

   On my Linux computer, the Terminal window responds with the following text:

```
java version 1.7.0_1
```

If your computer responds with the number 1.7.0 or higher, then you can pop open the champagne and look forward to some good times running this book's examples. If the version number is 1.5 or greater, then you can run many, but not all, of this book's examples. If your computer responds with something like `command not found`, then, most likely, Java isn't installed on your computer.

If you're juggling versions of Java on your computer . . .

I've had some good luck and some bad luck installing new versions of Java over old versions. If you can prepare your computer by uninstalling existing Java versions, you may save yourself some hassle in the long run. On the other hand, if uninstalling software makes you nervous, you can probably live with several Java versions on one computer. (As an author of Java books, I keep several versions of Java on my computer, and I seldom have any trouble with them. I created Figure 2-2 on a typical day with my Windows 7 computer.)

If you suspect that previously installed versions of Java are giving you trouble, the next few paragraphs describe some uninstallation tricks.

Uninstalling old Java versions on Windows XP

Select Start➪Control Panel➪Add or Remove Programs. Then, in the Add or Remove Programs dialog box, look for entries whose names begin with the word *Java*. (Typically, these entries have names like *Java 6* or *Java SE Development Kit 6*.)

When you highlight one of these entries, Windows displays a Remove button (see Figure 2-5). Click the Remove button and respond to the resulting prompts.

Uninstalling old Java versions on Windows 7

Select Start➪Control Panel➪Uninstall a Program. In the resulting list of installed programs, look for entries whose names begin with the word *Java*. (Typically, these entries have names like *Java 6* or *Java SE Development Kit 6*.)

When you right-click one of these entries, Windows displays a small context menu (see Figure 2-6). Select the Uninstall option in the context menu and then respond to the resulting prompts.

Uninstalling old Java versions on Windows 8

In the Start screen, select Control Panel➪More Settings➪Uninstall a Program. As a result, Windows displays a list of installed programs. From that point on, the steps to uninstall a program are the same as the Windows 7 steps.

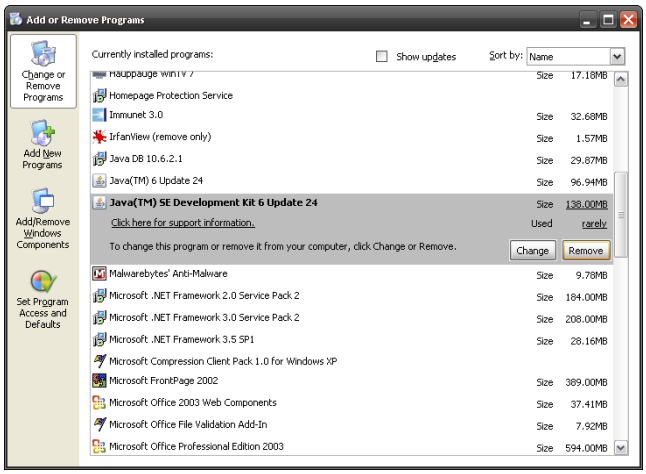

Add or Remove Programs
Currently installed programs:
Show updates
Sort by: Name
Change or Remove Programs
Add New Programs
Add/Remove Windows Components
Set Program Access and Defaults
Homepage Protection Service
Immunet 3.0
IrfanView (remove only)
Java DB 10.6.2.1
Java(TM) 6 Update 24
Java(TM) SE Development Kit 6 Update 24
Click here for support information.
To change this program or remove it from your computer, click Change or Remove.
Change
Remove
Malwarebytes' Anti-Malware
Microsoft .NET Framework 2.0 Service Pack 2
Microsoft .NET Framework 3.0 Service Pack 2
Microsoft .NET Framework 3.5 SP1
Microsoft Compression Client Pack 1.0 for Windows XP
Microsoft FrontPage 2002
Microsoft Office 2003 Web Components
Microsoft Office File Validation Add-In
Microsoft Office Professional Edition 2003

Figure 2-5: Uninstalling software on Windows XP.

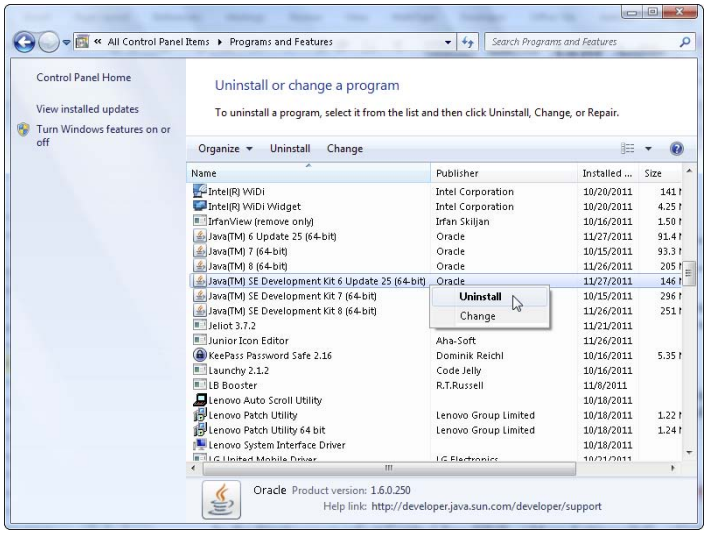


Figure 2-6: Uninstalling software on Windows 7.

Prioritizing Java versions on a Mac

Apple doesn't make uninstalling old Java versions easy. But you can manage installed versions of Java by running Mac's Java Preferences application. Here's what you do:

1. **In the Spotlight's search field, type** Java Preferences**.**
2. **When the Spotlight's Top Hit is Java Preferences, press Enter.**

   The Java Preferences window appears (refer to Figure 2-4).

3. **In the Java Applications list, drag your favorite version of Java (preferably Java 7 or later) to the top of the list.**
4. **Close the Java Preferences application.**

Uninstalling old Java versions on Linux

I'm sorry. Linux comes in too many flavors for me to list all the possibilities. To uninstall software on Linux, check the documentation specific to your Linux distribution.

Setting Up the Eclipse Integrated Development Environment

In the previous sections, you get all the tools your *computer* needs for processing Java programs. This section is different. In this section, you get the tool that *you* need for composing and testing your Java programs. You get Eclipse — an integrated development environment for Java.

An *integrated development environment* (IDE) is a program that provides tools to help you create software easily and efficiently. You can create Java programs without an IDE, but the time and effort you save using an IDE makes the IDE worthwhile. (Some hard-core programmers disagree with me, but that's another matter.)

According to the Eclipse Foundation's website, *Eclipse* is "a universal tool platform — an open extensible IDE for anything and nothing in particular." Indeed, Eclipse is versatile. Programmers generally think of Eclipse as an IDE for developing Java programs, but Eclipse has tools for programming in C++, in PHP, and in many other languages. I've even seen incarnations of Eclipse that have nothing to do with program development. (One such product is the *Lively Browser* — a web browser whose tabs are built from Eclipse components.)

Downloading Eclipse

Here's how you download Eclipse:

1. **Visit** `www.eclipse.org`**.**
2. **Look for a way to download Eclipse for your operating system.**

   Today, I visit `www.eclipse.org` and see a big button displaying the words Get Started Now . . . Download Eclipse (see Figure 2-7). Tomorrow, who knows what I'll see on this ever-changing website!

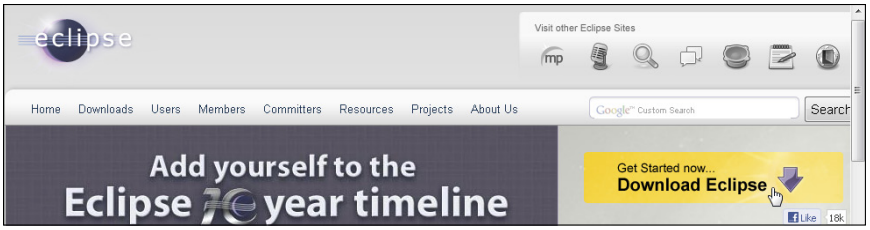


Figure 2-7: The home page for eclipse.org.

After clicking the Download Eclipse button, I see a list of downloads for my computer's operating system (see Figure 2-8).

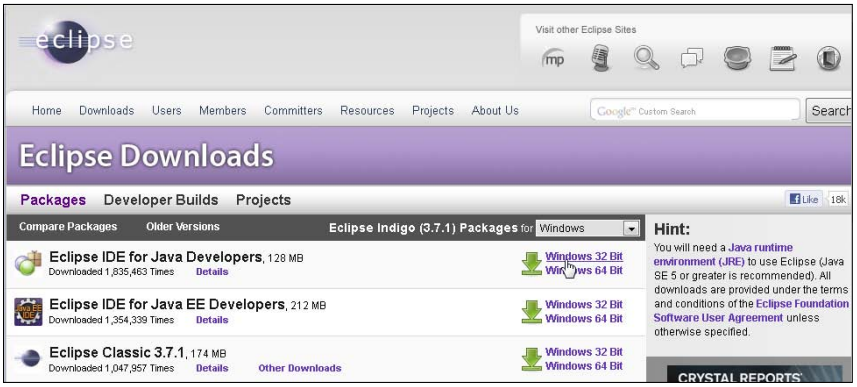

eclipse
Visit other Eclipse Sites
Home Downloads Users Members Committers Resources Projects About Us
Search
Eclipse Downloads
Packages Developer Builds Projects
Like 18k
Compare Packages Older Versions
Eclipse Indigo (3.7.1) Packages for Windows
Eclipse IDE for Java Developers, 128 MB
Downloaded 1,835,463 Times Details
Windows 32 Bit
Windows 64 Bit
Eclipse IDE for Java EE Developers, 212 MB
Downloaded 1,354,339 Times Details
Windows 32 Bit
Windows 64 Bit
Eclipse Classic 3.7.1, 174 MB
Downloaded 1,047,957 Times Details Other Downloads
Windows 32 Bit
Windows 64 Bit
Hint:
You will need a Java runtime environment (JRE) to use Eclipse (Java SE 5 or greater is recommended). All downloads are provided under the terms and conditions of the Eclipse Foundation Software User Agreement unless otherwise specified.
CRYSTAL REPORTS

Figure 2-8: Eclipse.org lists downloads for Windows.

Eclipse's download page directs you to versions of Eclipse that are specific to your computer's operating system. For example, if you visit the page on a Windows computer, the page shows you downloads for Windows only. If you're downloading Eclipse for use on another computer, you may want to override the automatic choice of operating system. Look for a little drop-down list containing the name of your computer's operating system. You can change the selected operating system in that drop-down list.

3. **Choose an Eclipse package from the available packages.**

   Regardless of your operating system, Eclipse comes in many shapes, sizes, and colors. The Eclipse website offers Eclipse IDE for Java Developers, Eclipse IDE for Java EE Developers, Eclipse Classic, and many other specialized downloads (see Figure 2-8). I usually select Eclipse IDE for Java Developers, and I recommend that you do the same.

4. **Choose between Eclipse's 32-bit and 64-bit versions.**

   If you know which Java version you have (32-bit or 64-bit), be sure to download the corresponding Eclipse version. If you don't know which Java version you have, download the 64-bit version of Eclipse and try

to launch it. If you can launch 64-bit Eclipse, you're okay. But if you get a `No Java virtual machine was found` error message, try downloading and launching the 32-bit version of Eclipse.

For the full lowdown on 32-bit and 64-bit word lengths, see this chapter's "How many bits does your computer have?" sidebar.

5. **Follow the appropriate links to get the download to begin.**

   The links you follow depend on which of Eclipse's many mirror sites is offering up your download. Just wade through the possibilities and get the download going.

Installing Eclipse

Precisely how you install Eclipse depends on your operating system and on what kind of file you get when you download Eclipse. Here's a brief summary:

- **If you run Windows and the download is an `.exe` file:**

  Double-click the `.exe` file's icon.

- **If you run Windows and the download is a `.zip` file:**

  Extract the file's contents to the directory of your choice.

  In other words, find the `.zip` file's icon in Windows Explorer. Then double-click the `.zip` file's icon. (As a result, Windows Explorer displays the contents of the `.zip` file, which consists of only one folder — a folder named `eclipse`.) Drag the `eclipse` folder to a convenient place in your computer's hard drive.

  For more information about `.zip` files, see the "Compressed archive files" sidebar in this chapter.

  My favorite place to drag the `eclipse` folder is directly onto the `C:` drive. So my `C:` drive has folders named `Program Files`, `Windows`, `eclipse`, and others. I avoid making the `eclipse` folder be a subfolder of `Program Files` because from time to time, I've had problems dealing with the blank space in the name `Program Files`.

- **If you run Mac OS X:**

  If you download a `.tar.gz` file, find the file in your Downloads folder and double-click it. Double-clicking the file should extract the file's contents. After extraction, your Downloads folder contains a new `eclipse` folder. Drag this new `eclipse` folder to your `Applications` folder, and you're all set.

  If you download a `.dmg` file, your web browser may open the file for you. If not, find the `.dmg` file in your Downloads folder and double-click the file. Follow any instructions that appear after this double-click. If you're expected to drag Eclipse into your Applications folder, do so.

If you run Linux:

You may get a `.tar.gz` file, but there's a chance you'll get a self-extracting `.bin` file. Extract the `.tar.gz` file to your favorite directory or execute the self-extracting `.bin` file.

Running Eclipse for the first time

The first time you launch Eclipse, you perform a few extra steps. To get Eclipse running, do the following:

1. **Launch Eclipse.**

   In Windows, the Start menu may not have an Eclipse icon. In that case, look in Windows Explorer for the folder containing your extracted Eclipse files. Double-click the icon representing the `eclipse.exe` file. (If you see an `eclipse` file but not an `eclipse.exe` file, check this chapter's "Those pesky filename extensions" sidebar.)

   On a Mac, go to the Spotlight and type `Eclipse` in the search field. When *Eclipse* appears as the Top Hit in the Spotlight's list, press Enter.

   When you launch Eclipse, you see a Workspace Launcher dialog (see Figure 2-9). The dialog asks where, on your computer's hard drive, you want to store the code that you will create using Eclipse.

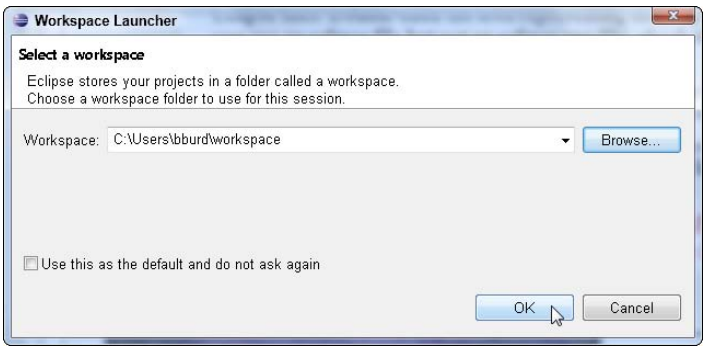


Figure 2-9: Eclipse's Workspace Launcher.

2. **In the Workspace Launcher dialog, click OK to accept the default (or don't accept the default!).**

   One way or another, it's no big deal!

   Because this is your first time using a particular Eclipse workspace, Eclipse starts with a Welcome screen (see Figure 2-10). Through the ages, most of the Eclipse Welcome screens have displayed icons along with little or no helpful text.

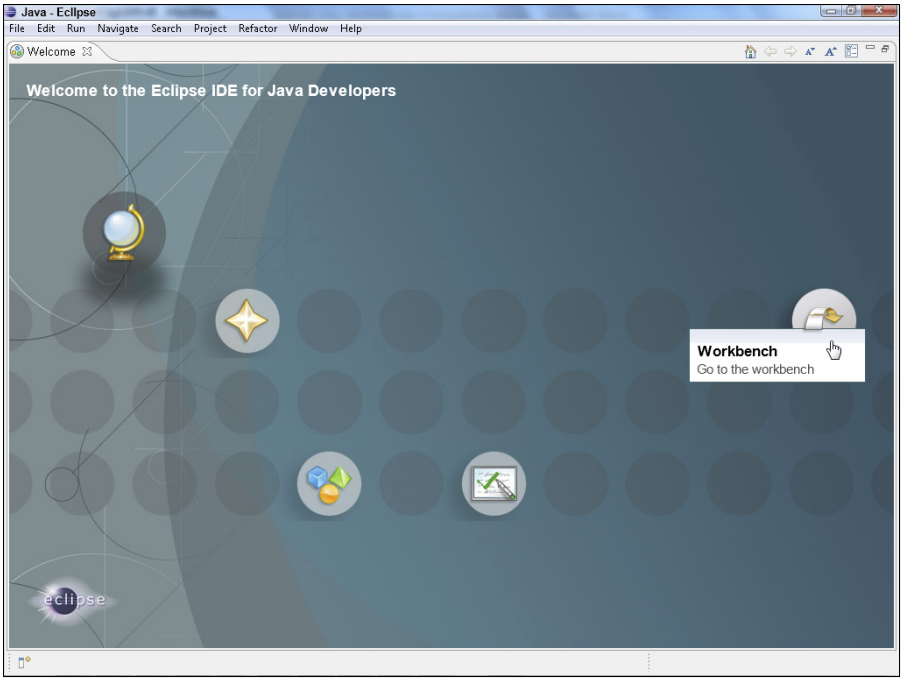


Figure 2-10: Eclipse's Welcome screen.

3. **Hover over the icons on Eclipse's Welcome screen until you find an icon whose tooltip contains the word Workbench.**
4. **Click the Workbench icon to open Eclipse's main screen.**

   A view of the main screen, after opening Eclipse with a brand-new workspace, is shown in Figure 2-11.

Configuring Java in Eclipse

Eclipse normally looks on your computer for Java installations and selects an installed version of Java to use for running your Java programs. Your computer may have more than one version of Java, so you might want to double-check Eclipse's choice of Java version. The following steps show you how:

1. **On Windows or Linux: In Eclipse's main menu, select Window➪ Preferences. On a Mac: In Eclipse's main menu, select Eclipse➪ Preferences.**

   As a result, Eclipse's Preferences dialog appears. (You can follow along with Figure 2-12.)
2. **In the tree on the left side of the Preferences dialog, expand the Java branch.**
3. **Within the Java branch, select the Installed JREs sub-branch.**

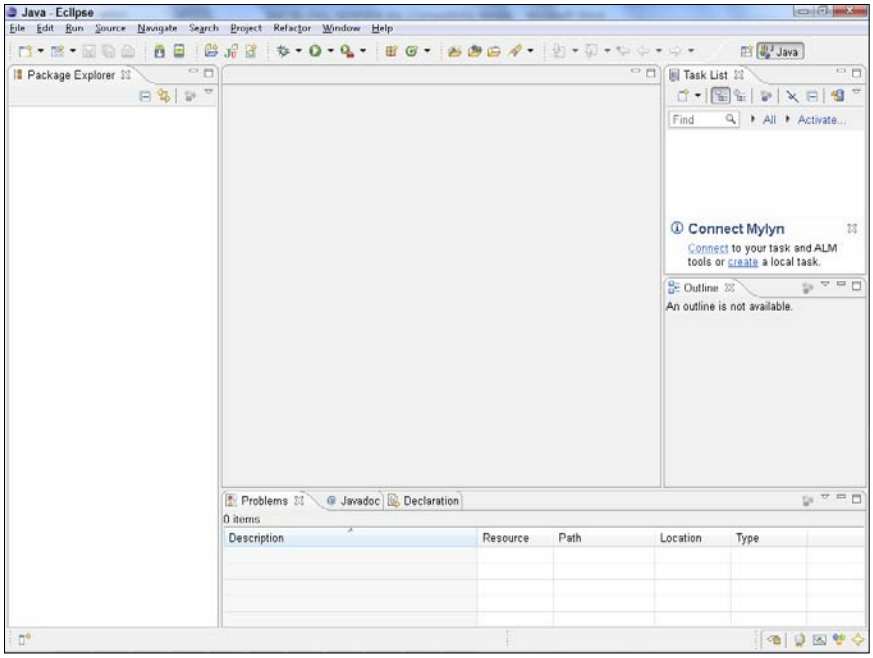


Figure 2-11: The Eclipse workbench with a brand-new workspace.

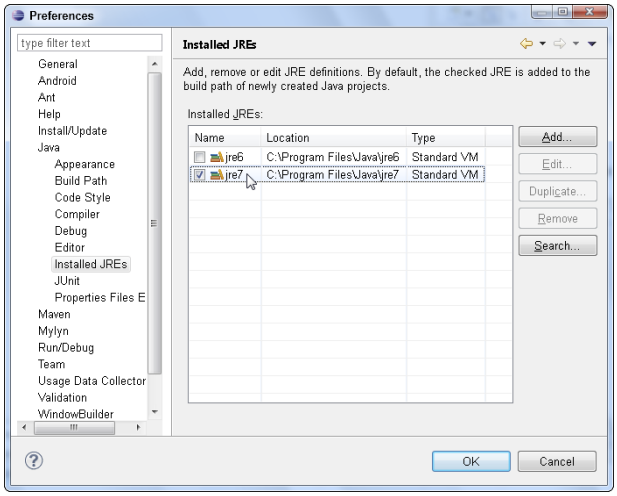


Figure 2-12: Eclipse's Preferences dialog.

4. **Look at the list of Java versions (Installed JREs) in the main body of the Preferences dialog.**

   In the list, each version of Java has a check box. Eclipse uses the version whose box is checked. If the checked version isn't your preferred version (for example, if the checked version isn't version 7 or higher), then you have to make some changes.

5. **If your preferred version of Java appears in the Installed JREs list, put a check mark in that version's check box.**
6. **If your preferred version of Java doesn't appear in the Installed JREs list, click the Add button.**

   When you click the Add button, a JRE Type dialog appears (see Figure 2-13).

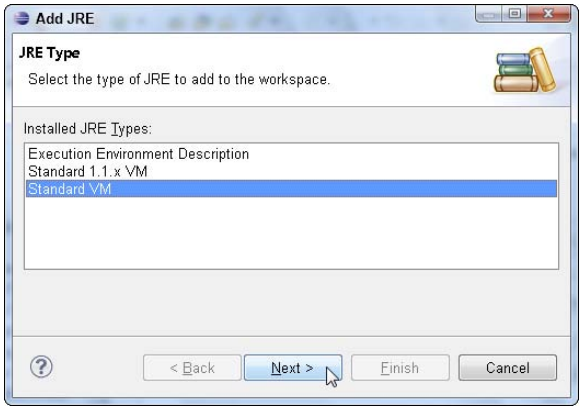


Figure 2-13: The JRE Type dialog.

7. **In the JRE Type dialog, double-click Standard VM.**

   As a result, a JRE Definition dialog appears (see Figure 2-14). What you do next depends on a few different things.

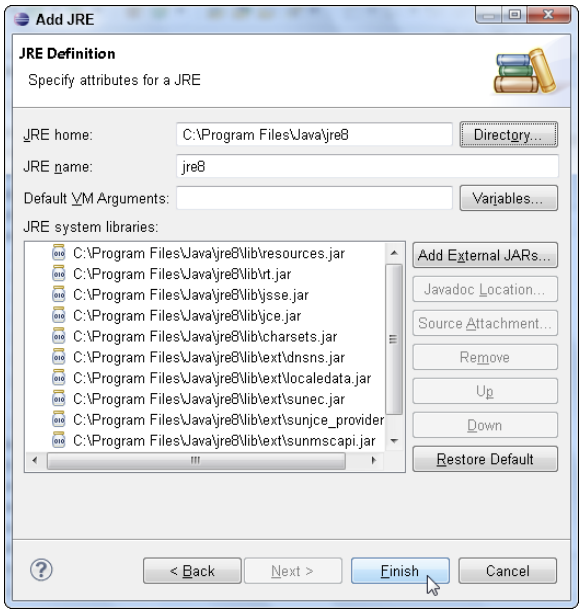


Figure 2-14: The JRE Definition dialog (after you've followed Steps 8 and 9).

8. **Fill in the dialog's JRE Home field.**

   How you do this depends on your operating system.

   - On Windows, browse to the directory in which you've installed your preferred Java version. On my many Windows computers, that directory is either `C:\Program Files\Java\jre7`, `C:\Program Files\Java\jdk1.7.0`, `C:\Program Files (x86)\Java\jre8` or something of that sort.
   - On a Mac, use the Finder to browse to the directory in which you've installed your preferred Java version. Type the name of the directory in the dialog's JRE Home field.

     My Mac has one Java directory named `/System/Library/Java/Java Virtual Machines/1.6.0jdk/Contents/Home` and another Java directory named `/Library/Java/JavaVirtualMachines/JDK 1.7.0 Developer Preview.jdk/Contents/Home`.

     Directories like `/System` and `/Library` don't normally appear in the Finder window. To browse to one of these directories (to the `/Library` directory, for example), choose Go⇨Go to Folder in the Finder's menu bar. In the resulting dialog, type `/Library` and then press Go.

     As you navigate toward the directory containing your preferred Java version, you might encounter a `JDK 1.7.0 Developer Preview.jdk` icon or some other item whose extension is `.jdk`. To see the contents of this item, control-click the items icon and then select Show Package Contents.
   - On Linux, browse to the directory in which you've installed your preferred Java version. When doubt, search for a directory whose name starts with `jre` or `jdk`.

   You might have one more thing to do back in the JRE Definition dialog.

9. **Look at the JRE Definition dialog's JRE Name field; if Eclipse hasn't filled in a name automatically, type a name (almost any text) in the JRE Name field.**

10. **Dismiss the JRE Definition dialog by clicking Finish.**

    Eclipse's Preferences dialog returns to the foreground. The box's Installed JREs list contains your newly added version of Java.

11. **Put a check mark in the check box next to your newly added version of Java.**

    You're almost done. (You have a few more steps to follow.)

12. **Within the Java branch on the left side of the Preferences dialog, select the Compiler sub-branch.**

    In the main body of the Preferences dialog, you see a Compiler Compliance Level drop-down list (see Figure 2-15).

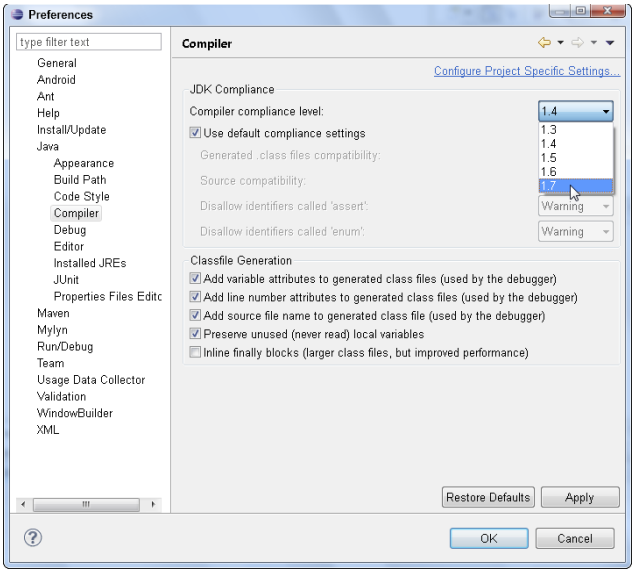


Figure 2-15: Setting the compiler compliance level.

13. **In the Compiler Compliance Level drop-down list, select a number that matches your preferred Java version.**

    For Java 7, I select compliance level 1.7. For Java 8, I select compliance level 1.8. (As I write this chapter, Level 1.8 isn't available in Eclipse. Java 8 has a preview release, but Eclipse won't have compliance level 1.8 for many months to come.)

14. **Whew! Click the Preferences dialog's OK button to return to the Eclipse workbench.**

Importing this book's sample programs

This import business can be tricky. As you move from one dialog to the next, many of the options have similar names. That's because Eclipse offers many different ways to import many different kinds of things. Anyway, if you follow these instructions, you'll be okay.

1. **Follow the steps in this chapter's earlier "Getting this Book's Sample Programs" section.**
2. **In Eclipse's main menu, choose File➪Import (see Figure 2-16).**

   As a result, Eclipse displays an Import dialog.

3. **In the Import dialog's tree, expand the General branch.**
4. **In the General branch, double-click the Existing Projects Into Workspace sub-branch (see Figure 2-17).**

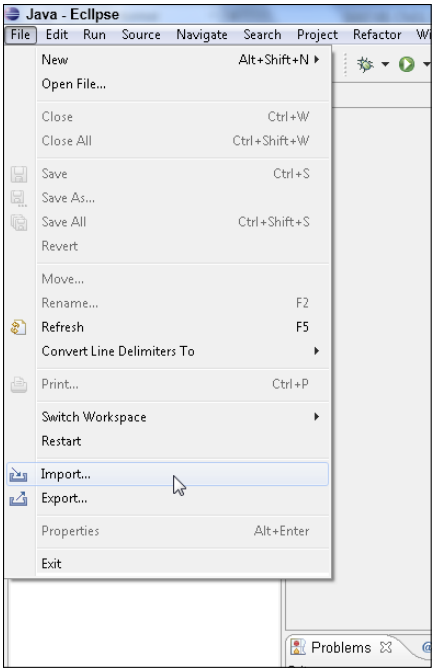


Figure 2-16: Starting to import this book's code.

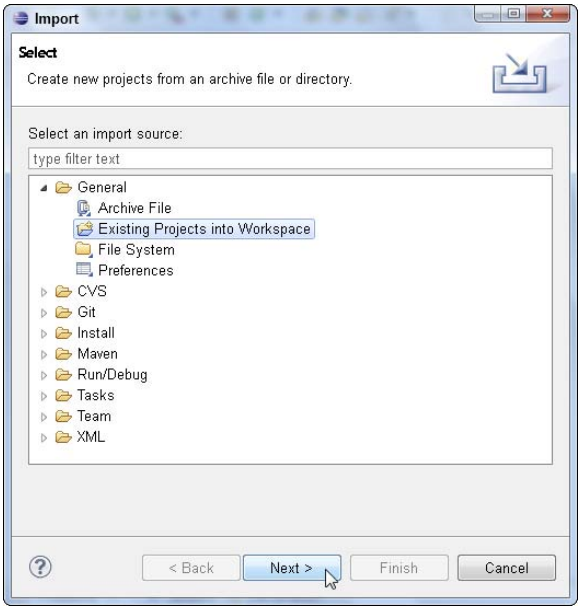


Figure 2-17: Among all the options, select Existing Projects into Workspace.

As a result, an Import Projects dialog appears.

5. **In the Import Projects dialog, select the Select Archive File radio button (see Figure 2-18).**

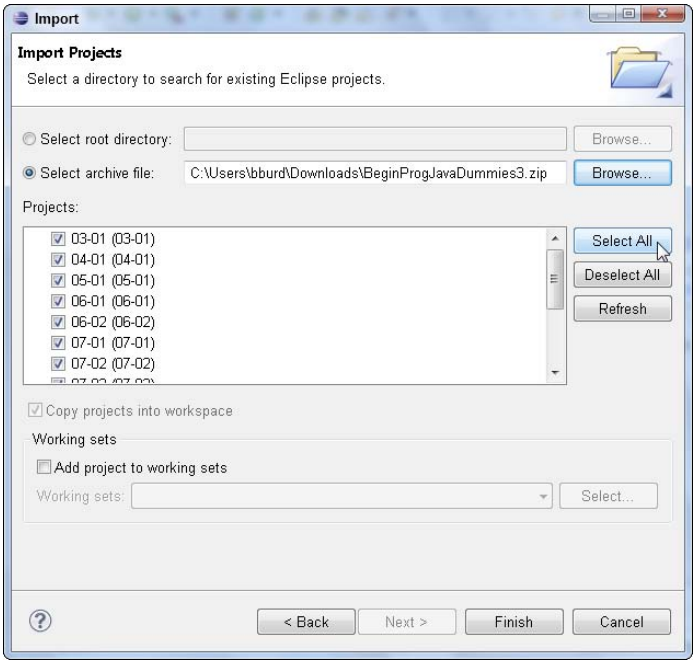


Figure 2-18: The Import Projects dialog.

This book's code lives in an archive file named `BeginProgJavaDummies3.zip`.

For the complete scoop on archive files, the sidebar entitled "Compressed archive files."

6. **Click the Browse button to find the `BeginProgJavaDummies3.zip` file on your computer's hard drive.**

   If you're not sure where to find the file, look first in a folder named `Downloads`.

   After you find the `BeginProgJavaDummies3.zip` file, Eclipse's Import Projects dialog displays the names of the projects inside the file. (Again, refer to Figure 2-18.)

7. **Click the Select All button.**

   This book's examples are so exciting that you want to import all of them!

8. **Click the Finish button.**

   As a result, the main Eclipse workbench reappears. The left side of the workbench displays the names of this book's Java projects (see Figure 2-19).

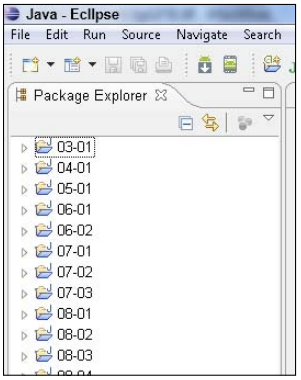


Figure 2-19: Eclipse displays a bunch of Java projects.

Now the real fun begins.

What's Next?

If you're reading this paragraph, you've probably followed some of the instructions in this chapter — instructions for installing Java and the Eclipse IDE on your computer. So the burning question is this: Have you done the installation correctly? The answer to that question lies in Chapter 3 because in that chapter, you use these tools to run a brand-new Java program.

Chapter 3

Running Programs

In This Chapter

- Compiling and running a program
- Working with a workspace
- Editing your own Java code

If you're a programming newbie, for you, running a program probably means clicking a mouse. You want to run Internet Explorer, so you double-click the Internet Explorer icon. That's all there is to it.

When you create your own programs, the situation is a bit different. With a new program, the programmer (or someone from the programmer's company) creates the program's icon. Before that process, a perfectly good program may not have an icon at all. So what do you do with a brand-new Java program? How do you get the program to run? This chapter tells you what you need to know.

Running a Canned Java Program

The best way to get to know Java is to do Java. When you're doing Java, you're writing, testing, and running your own Java programs. This section prepares you by describing how you run and test a program. Instead of writing your own program, you run a program that I've already written for you. The program calculates your monthly payments on a home mortgage loan.

The mortgage-calculating program doesn't open its own window. Instead, the program runs in Eclipse's Console view. The Console view is one of the tabs in the lower-right part of the Eclipse workbench (see Figure 3-1). A program that operates completely in this Console view is called a *text-based program.*

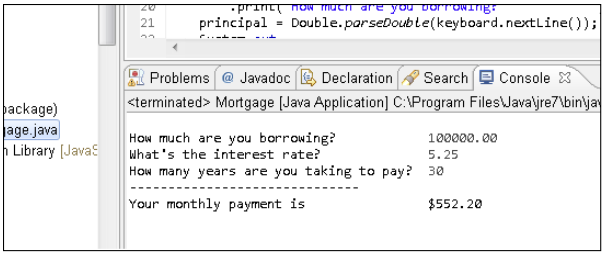


Figure 3-1: A run of this chapter's text–based mortgage program.

You may not see a Console tab in the lower-right part of the Eclipse workbench. To coax the Console view out of hiding, choose Window➪Show View➪Other. In the resulting Show View dialog box, expand the General branch. Finally, within that General branch, double-click the Console item.

For more information about the Console view (and about Eclipse's workbench in general), see the "Views and editors" section, later in this chapter.

Actually, as you run the mortgage program, you see two things in Eclipse's Console view:

- **Messages and results that the mortgage program sends to you.** Messages include things like `How much are you borrowing?` Results include lines like `Your monthly payment is $552.20`.
- **Responses that you give to the mortgage program while it runs.** If you type 100000.00 in response to the program's question about how much you're borrowing, you see that number echoed in Eclipse's Console view.

Here's how you run the mortgage program:

1. **Make sure that you've followed the instructions in Chapter 2 — instructions for installing Java, for installing and configuring Eclipse, and for getting this book's sample programs.**

   Thank goodness! You don't have to follow those instructions more than once.

2. **Launch Eclipse.**

   The Eclipse Workspace Launcher dialog box appears (see Figure 3-2).

   A *workspace* is a folder on your computer's hard drive. Eclipse stores your Java programs in one or more workspace folders. Along with these Java programs, each workspace folder contains some Eclipse settings. These settings store things like the version of Java that you're using, the colors that you prefer for words in the editor, the size of the editor area when you drag the area's edges, and other things. You can have several workspaces with different programs and different settings in each workspace.

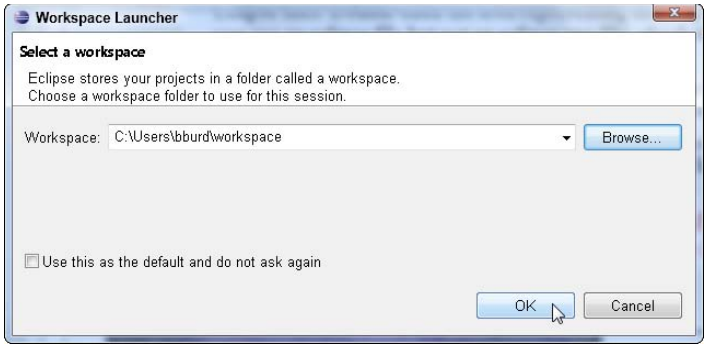


Figure 3-2: The Eclipse Workspace Launcher.

By default, the Workspace Launcher offers to open whatever workspace you opened the last time you ran Eclipse. You want to open the workspace that you used in Chapter 2, so don't modify the stuff in the Launcher's Workspace field.

3. **In the Workspace Launcher dialog box, click OK.**

   The big Eclipse workbench stares at you from your computer screen (see Figure 3-3).

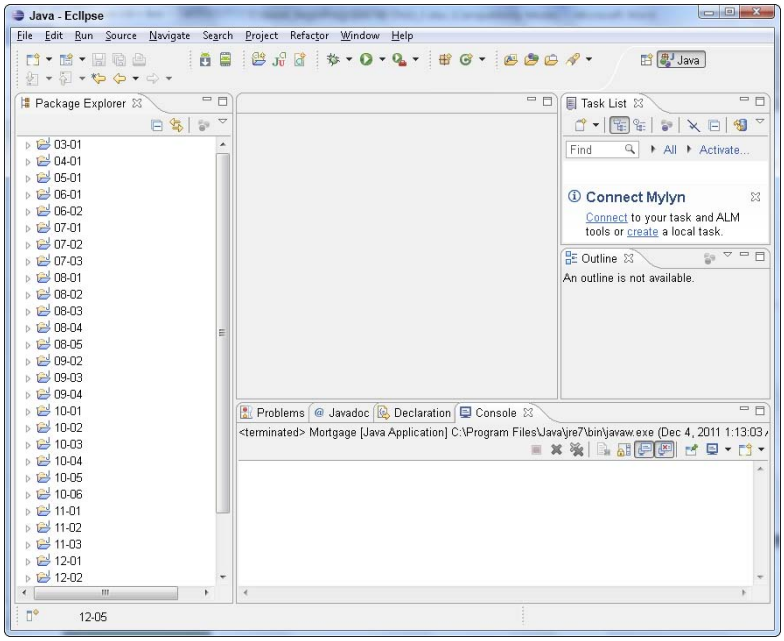


Figure 3-3: The Eclipse workbench.

In Figure 3-3, the leftmost part of the workbench is Eclipse's Package Explorer, which contains numbers like 03-01, 04-01, and so on. Each number is actually the name of an Eclipse *project*. Formally, a project is a collection of files and folders inside a workspace. Intuitively, a project is a basic work unit. For example, a self-contained collection of Java program files to manage your CD collection (along with the files containing the data) may constitute a single Eclipse project.

Looking again at the Package Explorer in Figure 3-3, you see projects named 03-01, 04-01, and so on. My project 03-01 holds the first and only example in Chapter 3 (this chapter). Project 06-02 contains the Java program in Listing 6-2 (the second code listing in Chapter 6 of this book). Project names can include letters, digits, blank spaces, and other characters, but for the names of this book's examples, I stick with digits and dashes.

To read more about things like Eclipse's Package Explorer, see the upcoming section "What's All That Stuff in Eclipse's Window?"

When you launch Eclipse, you may see something different from the stuff in Figure 3-3. You may see Eclipse's Welcome screen with only a few icons in an otherwise barren window. You may also see a workbench like the one in Figure 3-3, but without a list of numbers (03-01, 04-01, and so on) in the Package Explorer. If so, then you may have missed some instructions on configuring Eclipse in Chapter 2. Alternatively, you may have modified the stuff in the Launcher's Workspace field in Step 2 of this section's instructions.

In any case, make sure that you see numbers like 03-01 and 04-01 in the Package Explorer. Seeing these numbers assures you that Eclipse is ready to run the sample programs from this book.

4. **In the Package Explorer, click the `03-01` branch.**

   This chapter's Java project — the `03-01` project — appears highlighted.

   You may want to see a sneak preview of some Java code. To see the Java program that you're running in `Project 03-01`, expand the `03-01` branch in the Package Explorer. Inside the `03-01` branch, you find a `src` branch, which in turn contains a `(default package)` branch. Inside the `(default package)` branch, you find the `Mortgage.java` branch. That `Mortgage.java` branch represents my Java program. Double-clicking the `Mortgage.java` branch makes my code appear in Eclipse's editor.

5. **Choose Run➪Run As➪Java Application from the main menu, as shown in Figure 3-4.**

   When you choose Run As➪Java Application, the computer runs the project's code. (In this example, the computer runs a Java program that I wrote.) As part of the run, the message `How much are you borrowing?` appears in Eclipse's Console view. (The Console view shares

the lower-right area of Eclipse's workbench with the Problems view, the Javadoc view, the Declaration view, and possibly other views. Refer to Figure 3-1.)

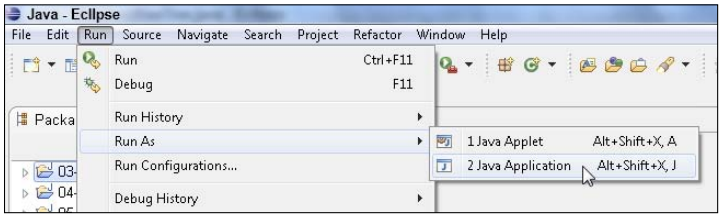


Figure 3-4: One of the ways to run the code in `Project 03-01`.

6. **Click anywhere inside Eclipse's Console view and then type a number, like 100000.00, and press Enter.**

When you type a number in Step 6, don't include your country's currency symbol and don't group the digits. (U.S. residents, don't type a dollar sign and don't use any commas.) Things like $100000.00 and 1,000,000.00 cause the program to crash. You see a `NumberFormatException` message in the Console view.

After you press Enter, the Java program displays another message (`What's the interest rate?`) in the Console view. (Again, refer to Figure 3-1.)

7. **In response to the interest rate question, type a number, like 5.25, and press Enter.**

After you press Enter, the Java program displays another message (`How many years . . . ?`) in the Console view.

8. **Type a number, like 30, and press Enter.**

In response to the numbers that you type, the Java program displays a monthly payment amount. Again, refer to Figure 3-1.

Disclaimer: Your local mortgage company charges more than the amount that my Java program calculates. (A lot more.)

When you type a number in Step 8, don't include a decimal point. Numbers like 30.0 cause the program to crash. You see a `NumberFormatException` message in the Console view.

Occasionally, you decide in the middle of a program's run that you've made a mistake of some kind. You want to stop the program's run dead in its tracks. Simply click the little red rectangle above the Console view (see Figure 3-5).

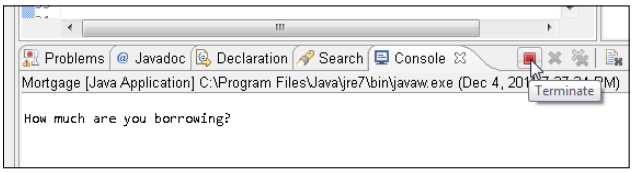


Figure 3-5: How to prematurely terminate a program's run.

If you follow this section's instructions and you don't get the results that I describe, you can try three things. I list them in order from best to worst:

- Check all the steps to make sure that you did everything correctly.
- Send e-mail to me at `BeginProg@allmycode.com`. If you describe what happened, I can probably figure out what went wrong and tell you how to correct the problem.
- Panic.

Typing and Running Your Own Code

The previous section is about running someone else's Java code (code that you download from this book's website). But eventually, you'll write code on your own. This section shows you how to create code with the Eclipse IDE.

Separating your programs from mine

You can separate your code from this book's examples by creating a separate workspace. Here are two ways to do it:

- **When you launch Eclipse, type a new folder name in the Workspace field of Eclipse's Workspace Launcher dialog box.**

  If the folder doesn't already exist, Eclipse creates the folder. If the folder already exists, Eclipse's Package Explorer lists any projects that the folder contains.

- **In the Eclipse workbench's main menu, choose File➪Switch Workspace (see Figure 3-6).**

  When you choose File➪Switch Workspace, Eclipse offers you a few of your previously opened workspace folders. If your choice of folder isn't in the list, select the Other option. In response, Eclipse reopens its Workspace Launcher dialog box.

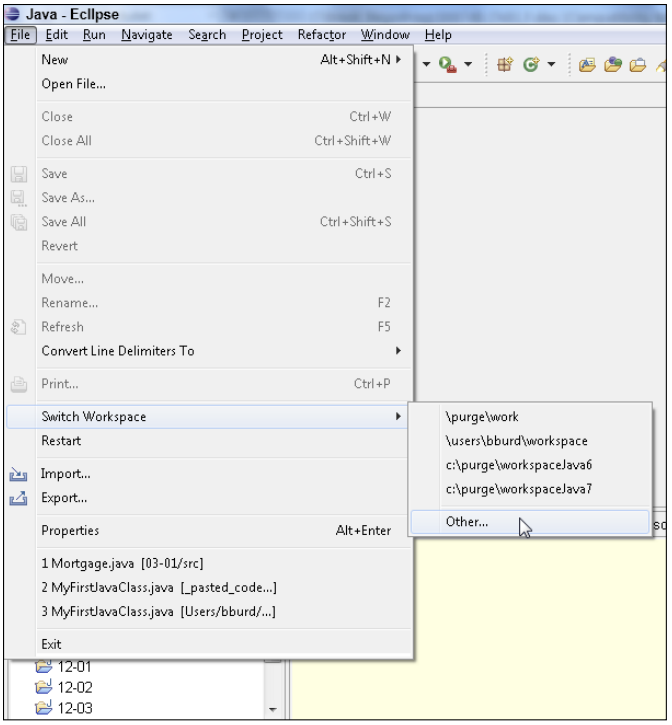


Figure 3-6: Switching to a different Eclipse workspace.

Writing and running your program

Here's how you create a new Java project:

1. **Launch Eclipse.**
2. **From Eclipse's menu bar, choose File➪New➪Java Project.**

   A New Java Project dialog box appears.
3. **In the New Java Project dialog box, type a name for your project and then click Finish.**

   In Figure 3-7, I type the name `MyFirstProject`.

   If you click Next instead of Finish, you see some other options that you don't need right now. So to avoid any confusion, just click Finish.

   Clicking Finish brings you back to Eclipse's workbench, with `MyFirstProject` in the Package Explorer, as shown in Figure 3-8.

   The next step is to create a new Java source code file.

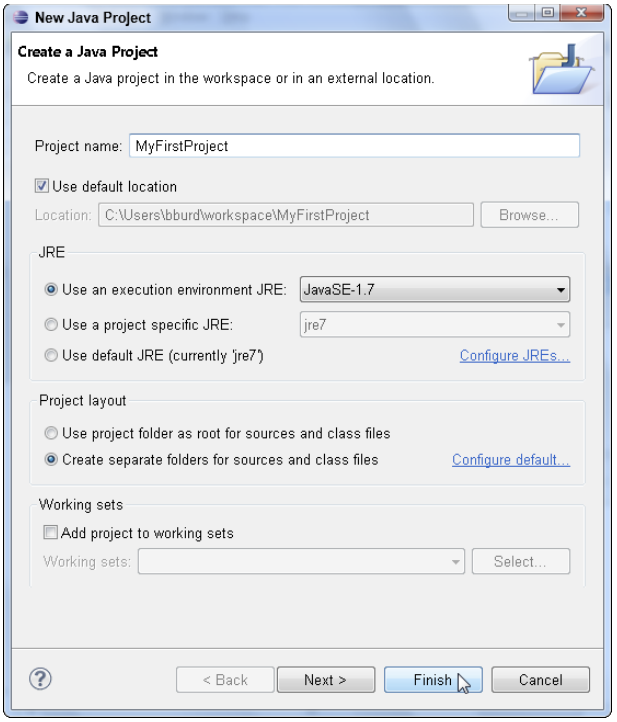


Figure 3-7: Getting Eclipse to create a new project.

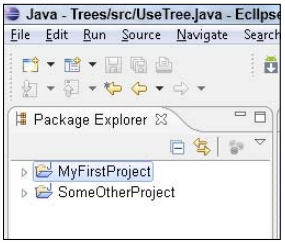


Figure 3-8: Your project appears in Eclipse's Package Explorer.

4. **Select your newly created project in the Package Explorer.**

   To create Figure 3-8, I selected `MyFirstProject` instead of `SomeOtherProject`.

5. **In Eclipse's main menu, choose File➪New➪Class.**

   Eclipse's New Java Class dialog box appears (see Figure 3-9).

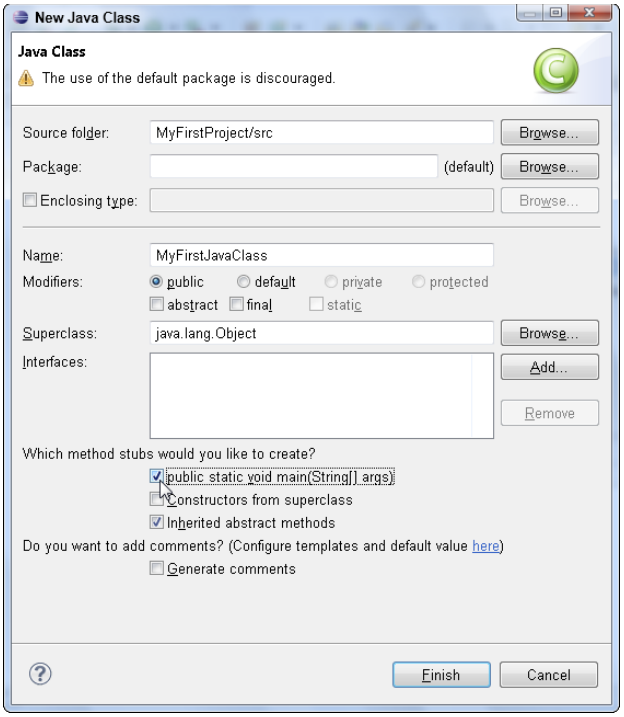


Figure 3-9: Getting Eclipse to create a new Java class.

Java programmers normally divide their code into one or more *packages*. A typical package has a name like `java.util` or `com.allmycode.images`. In Figure 3-9, Eclipse is warning me that I'm not naming a package to contain my project's code. So the code goes into a nondescript thing called Java's *default package*. Java's default package is a package with no name — a catchall location for code that isn't otherwise packaged. Packages are great for managing big programming projects, but *Beginning Programming For Dummies* contains no big programming projects. So in this example (and in all of this book's examples), I choose to ignore the warning. For more info about Java packages, see Chapter 18.

Like every other windowed environment, Eclipse provides many ways to accomplish the same task. Instead of choosing File➪New➪Class, you can right-click `MyFirstProject` in the Package Explorer in Windows (or control-click `MyFirstProject` in the Package Explorer on a Mac). In the resulting context menu, choose New➪Class. You can also start by pressing Alt+Shift+N in Windows (or Option+Cmd+N on a Mac). The choice of clicks and keystrokes is up to you.

Do I see formatting in my Java program?

When you use Eclipse's editor to write a Java program, you see words in various colors. Certain words are always blue. Other words are always black. You even see some bold and italic phrases. You may think you see formatting, but you don't. Instead, what you see is called *syntax coloring* or *syntax highlighting*.

No matter what you call it, the issue is as follows:

- With Microsoft Word, things like bold formatting are marked inside a document. When you save `MyPersonalDiary.doc`, the instructions to make the words "love" and "hate" bold are recorded inside the `MyPersonalDiary.doc` file.
- With a Java program editor, things like bold and coloring aren't marked inside the Java program file. Instead, the editor displays each word in a way that makes the Java program easy to read.

For example, in a Java program, certain words (words like `class`, `public`, and `void`) have their own special meanings. So Eclipse's editor displays `class`, `public`, and `void` in bold, reddish letters. When I save my Java program file, the computer stores nothing about bold, colored letters in my Java program file. But the editor uses its discretion to highlight special words with reddish coloring.

Some other editor may display the same words in a blue font. Another editor (like Windows Notepad) displays all words in plain old black.

6. **In the New Java Class dialog box's Name field, type the name of your new class.**

   In this example, I use the name `MyFirstJavaClass`, with no blank spaces between any of the words in the name. (Refer to Figure 3-9.)

   The name in the New Java Class dialog box must not have blank spaces. And the only allowable punctuation symbol is the underscore character (_). You can name your class `MyFirstJavaClass` or `My_First_Java_Class`, but you can't name it `My First Java Class` or `JavaClass,MyFirst`.

7. **Put a check mark in the `public static void main(String[] args)` check box.**

   Your check mark tells Eclipse to create some boilerplate Java code.

8. **Accept the defaults for everything else in the New Java Class dialog box (in other words, click Finish).**

   You can even ignore the "default package is discouraged" warning near the top of the dialog box.

   Clicking Finish brings you back to Eclipse's workbench. Now `MyFirstProject` contains a file named `MyFirstJavaClass.java`. For your convenience, the `MyFirstJavaClass.java` file already has some code in it. Eclipse's editor displays the Java code (see Figure 3-10).

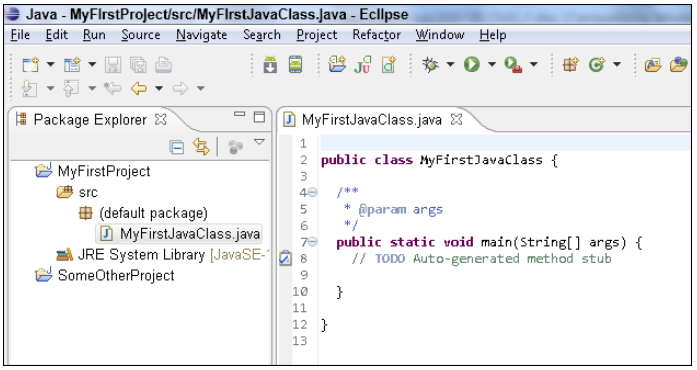


Figure 3-10: Eclipse writes some code in the Editor.

9. **Replace an existing line of code in your new Java program.**

   Type a line of code in Eclipse's Editor. Replace the line

   ```
   // TODO Auto-generated method stub
   ```

   with the line

   ```
   System.out.println("Chocolate, royalties, sleep");
   ```

   Copy the new line of code exactly as you see it in Listing 3-1.

   - Spell each word exactly the way I spell it in Listing 3-1.
   - Capitalize each word exactly the way I do in Listing 3-1.
   - Include all the punctuation symbols — the dots, the quotation marks, the semicolon, everything.
   - Distinguish between the lowercase letter `l` and the digit `1`. The word `println` tells the computer to *print* a whole *line*. Each character in the word `println` is a lowercase letter. The word contains no digits.

Listing 3-1: A Program to Display the Things I Like

```
public class MyFirstJavaClass {

  /**
   * @param args
   */
  public static void main(String[] args) {
    System.out.println("Chocolate, royalties, sleep");

  }

}
```

Java is *case-sensitive*, which means that `system.out.printLn` isn't the same as `System.out.println`. If yOu tyPe `system.out.printLn`, your prograAm won't worK. Be sUre to cAPItalize your codE eXactLy as it is in LiSTIng 3-1.

If you typed everything correctly, you see the stuff in Figure 3-11.

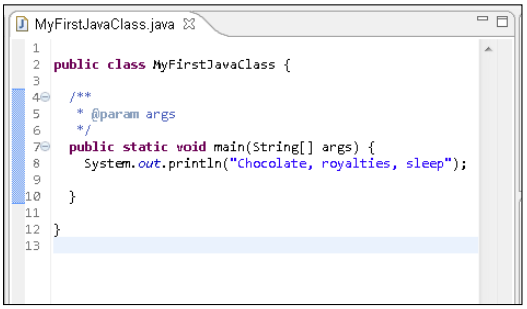


Figure 3-11: A Java program in the Eclipse editor.

If you don't type the code exactly as it's shown in Listing 3-1, you may see jagged red underlines, tiny rectangles with X-like markings inside them, or other red marks in the Editor (see Figure 3-12).

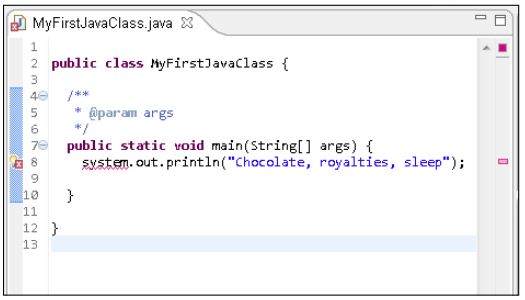


Figure 3-12: A Java program typed incorrectly.

The red marks in Eclipse's editor refer to *compile-time errors* in your Java code. A compile-time error (also known as a *compiler error*) is an error that prevents the computer from translating your code. (See the talk about code translation in Chapter 1.)

The error markers in Figure 3-12 appear on line 8 of the Java program. Line numbers appear in the editor's left margin. To make Eclipse's editor display line numbers, choose Window➪Preferences (on Windows) or Eclipse➪Preferences (on a Mac). Then choose General➪Editors➪Text Editors. Finally, put a check mark in the Show Line Numbers check box.

To fix compile-time errors, you must become a dedicated detective. You join an elite squad known as *Law & Order: Java Programming Unit*. You

seldom find easy answers. Instead, you comb the evidence slowly and carefully for clues. You compare everything you see in the editor, character by character, with my code in Listing 3-1. You don't miss a single detail, including spelling, punctuation, and uppercase versus lowercase.

Eclipse has a few nice features to help you find the source of a compile-time error. For example, you can hover over the jagged red underline. When you do, you see a brief explanation of the error along with some suggestions for repairing the error — some *quick fixes* (see Figure 3-13).

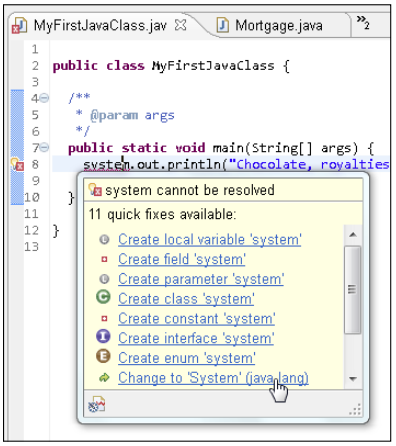


Figure 3-13: Eclipse offers some helpful suggestions.

In Figure 3-13, a popup tells you that Java doesn't know what the word *system* means — that is, *system cannot be resolved*. Near the bottom of the figure, one of the quick fix options is to change *system* to *System*.

When you click that Change To 'System' (`java.lang`) option, Eclipse's editor replaces *system* with *System*. The editor's error markers disappear, and you go from the incorrect code in Figure 3-12 to the correct code in Figure 3-11.

10. **Make any changes or corrections to the code in the Eclipse's editor.**

    When at last you see no jagged underlines or blotches in the editor, you're ready to try running the program.

11. **Select `MyFirstJavaClass` either by clicking inside the editor or by clicking the `MyFirstProject` branch in the Package Explorer.**

12. **In Eclipse's main menu, choose Run➪Run As➪Java Application.**

    That does the trick. Your new Java program runs in Eclipse's Console view. If you're running the code in Listing 3-1, you see the `Chocolate, royalties, sleep` message in Figure 3-14. It's like being in heaven!

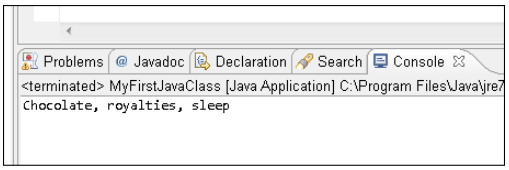


Figure 3-14: Running the program in Listing 3-1.

What can possibly go wrong?

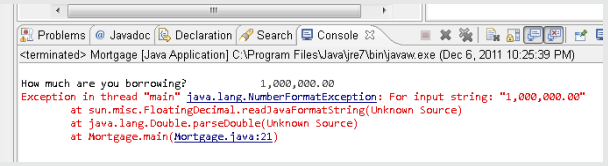
Ridding the editor of jagged underlines is cause for celebration. Eclipse likes the look of your code, so from that point on, it's smooth sailing. Right?

Well, it ain't necessarily so. In addition to some conspicuous compile-time errors, your code can have other, less obvious errors.

Imagine someone telling you to "go to the intersection, and then *rurn tight*." You notice immediately that the speaker made a mistake, and you respond with a polite "Huh?" The nonsensical *rurn tight* phrase is like a compile-time error. Your "Huh?" is like the jagged underlines in Eclipse's editor. As a listening human being, you may be able to guess what *rurn tight* means, but Eclipse's editor never dares to fix your code's mistakes.

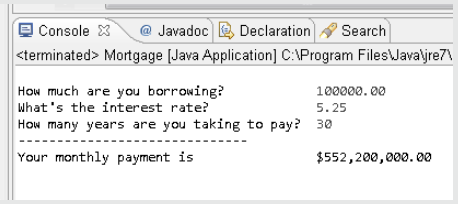
In addition to compile-time errors, some other kinds of gremlins can hide inside a Java program:

- **Unchecked runtime exceptions:** You have no compile-time errors, but when you run your program, the run ends prematurely. Somewhere in the middle of the run, your instructions tell Java to do something that can't be done. For example, while you're running the Mortgage program in the "Running a Canned Java Program" section, you type 1,000,000.00 instead of 1000000.00. Java doesn't like the commas in the number, so your program crashes and displays a nasty-looking message, as shown in the figure.



  This is an example of an *unchecked runtime exception* — the equivalent of someone telling you to turn right at the intersection when the only thing to the right is a big brick wall. Eclipse's editor doesn't warn you about an unchecked runtime exception because, until you run the program, the computer can't predict that the exception will occur.

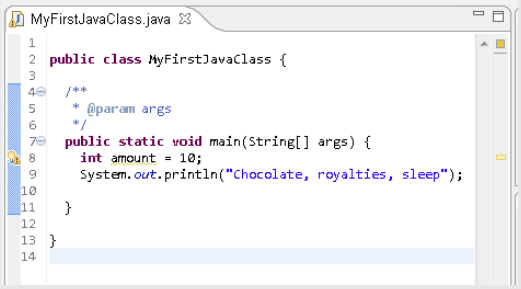
- **Logic errors:** You see no error markers in Eclipse's editor, and when you run your code, the program runs to completion. But the answer isn't correct. Instead of $552.20 in the figure, the output is $552,200,000.00. The program wrongly tells you to pay thousands of times what your house is worth and tells you to pay this amount each month! It's the equivalent of being told to turn right instead of turning left. You can drive in the wrong direction for a very long time.



Logic errors are the most challenging errors to find and to fix. And worst of all, logic errors often go unnoticed. In March 1985, I got a monthly home heating bill for $1,328,932.21. Clearly, some computer had printed the incorrect amount. When I called the gas company to complain about it, the telephone service representative said, "Don't be upset. Pay only half that amount."

- **Compile-time warnings:** A warning isn't as severe as an error message. So, when Eclipse notices something suspicious in your program, the editor displays a jagged yellow underline, a tiny yellow icon containing an exclamation point, and a few other not-so-intrusive clues.

  For example, in the figure on line 8, I add something about `amount = 10` to the code from Listing 3-1. The problem is, I never make use of the `amount` or of the number `10` anywhere in my program. With its faint yellow markings, Eclipse effectively tells me "Your `amount = 10` code isn't bad enough to be show stopper. Eclipse can still manage to run your program. But are you sure you want `amount = 10` (this stuff that seems to serve no purpose) in your program?"



Imagine being told to "turn when you reach the intersection." The direction may be just fine. But if you're suspicious, you ask, "Which way should I turn? Left or right?"

When you're sure that you know what you're doing, you can ignore warnings and worry about them at some later time. But a warning can be an indicator that something more serious is wrong with your code. So my sweeping recommendation is this: Pay attention to warnings. But if you can't figure out why you're getting a particular warning, don't let the warning prevent you from moving forward.

What's All That Stuff in Eclipse's Window?

Believe it or not, an editor once rejected one of my book proposals. In the margins, the editor scribbled "This is not a word" next to things like "can't," "it's," and "I've." To this day, I still do not know what this editor did not like about contractions. My own opinion is that language always needs to expand. Where would we be without a new words — words like *dotcom*, *infomercial*, and *vaporware*?

Even the *Oxford English Dictionary* (the last word in any argument about words) grows by more than 4,000 entries each year. That's an increase of more than 1 percent per year. It's about 11 new words per day!

The fact is, human thought is like a big high-rise building. You can't build the 50th floor until you've built at least part of the 49th. You can't talk about *spam* until you have a word like *e-mail*. With all that goes on these days, you need verbal building blocks. That's why this section contains a bunch of new terms.

In this section, each newly defined term describes an aspect of the Eclipse IDE. So before your read all this Eclipse terminology, I provide the following disclaimers:

- **This section is optional reading.** Refer to this section if you have trouble understanding some of this book's instructions. But if have no trouble navigating the Eclipse IDE, don't complicate things by fussing over the terminology in this section.
- **This section provides explanations of terms, not formal definitions of terms.** Yes, my explanations are fairly precise, but no, they're not airtight. Almost every description in this section has hidden exceptions, omissions, exemptions, and exclusions. Take the paragraphs in this section to be friendly reminders, not legal contracts.
- **Eclipse is a very useful tool.** But Eclipse isn't officially part of the Java ecosystem. Although I don't describe details in this book, you can write Java programs without ever using Eclipse.

Understanding the big picture

Your tour of Eclipse begins with a big Burd's eye view.

- **Workbench:** The Eclipse desktop (see Figure 3-3). The workbench is the environment in which you develop code.
- **Area:** A section of the workbench. The workbench in Figure 3-3 contains five areas. To illustrate the point, I've drawn borders around each of the areas (see Figure 3-15).

- **Window:** A copy of the Eclipse workbench. With Eclipse, you can have several copies of the workbench open at once. Each copy appears in its own window.

  To open a second window, go to the main Eclipse menu bar and choose Window➪New Window.

- **Action:** A choice that's offered to you, typically when you click something. For example, when you choose File➪New in Eclipse's main menu bar, you see a list of new things that you can create. The list usually includes Project, Folder, File, and Other, but it may also include things like Package, Class, and Interface. Each of these things (each item in the menu) is called an *action*.

Views, editors, and other stuff

The next bunch of terms deals with things called views, editors, and tabs.

You may have difficulty understanding the difference between views and editors. (A view is like an editor, which is like a view, or something like that.) If views and editors seem the same to you, and you're not sure you can tell which is which, don't be upset. As an ordinary Eclipse user, the distinction between views and editors comes naturally as you gain experience using the workbench. You rarely have to decide whether the thing you're using is a view or an editor.

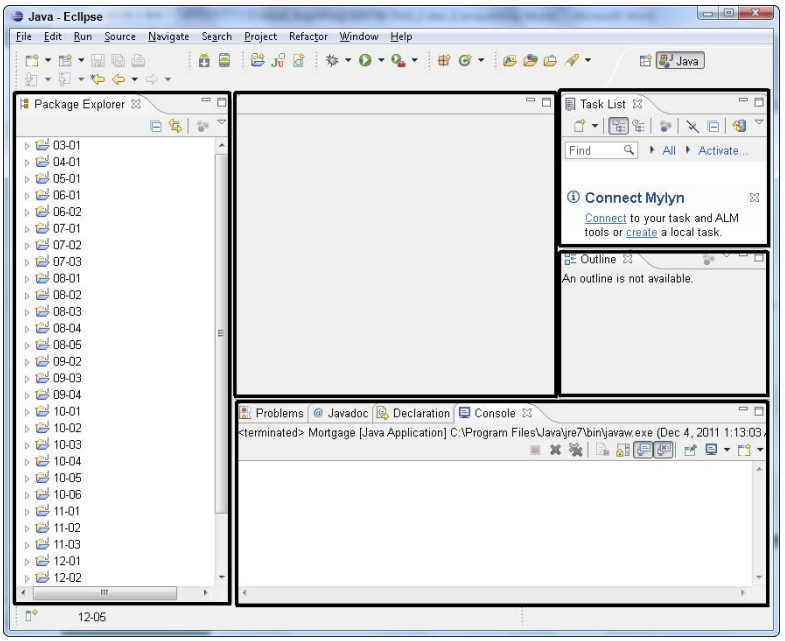


Figure 3-15: The workbench is divided into areas.

If you ever have to decide what a view is as opposed to an editor, here's what you need to know:

- **View:** A part of the Eclipse workbench that displays information for you to browse. In the simplest case, a view fills up an area in the workbench. For example, in Figure 3-3, the Package Explorer view fills up the left-most area.

  Many views display information as lists or trees. For example, in Figure 3-10, the Package Explorer view contains a tree.

  You can use a view to make changes to things. For example, to delete `SomeOtherProject` in Figure 3-10, right-click the `SomeOtherProject` branch in the Package Explorer view. (On a Mac, control-click the `SomeOtherProject` branch.) Then, in the resulting context menu, choose Delete.

  When you use a view to change something, the change takes place immediately. For example, when you choose Delete in the Package Explorer's context menu, whatever item you've selected is deleted immediately. In a way, this behavior is nothing new. The same kind of thing happens when you recycle a file using Windows Explorer or trash a file using the Macintosh Finder.

- **Editor:** A part of the Eclipse workbench that displays information for you to modify. A typical editor displays information in the form of text. This text can be the contents of a file. For example, an editor in Figure 3-10 displays the contents of the `MyFirstJavaClass.java` file.

  When you use an editor to change something, the change doesn't take place immediately. For example, look at the editor in Figure 3-10. This editor displays the contents of the `MyFirstJavaClass.java` file. You can type all kinds of things in the editor. Nothing happens to `MyFirstJavaClass.java` until you choose File➪Save from Eclipse's menu bar. Of course, this behavior is nothing new. The same kind of thing happens when you work in Microsoft Word or in any other word processing program.

  Like other authors, I occasionally become lazy and use the word "view" when I really mean "view or editor." When you catch me doing this, just shake your head and move onward. When I'm being very careful, I use the official Eclipse terminology. I refer to views and editors as *parts* of the Eclipse workbench. Unfortunately, this "parts" terminology doesn't stick in peoples' minds very well.

An area of the Eclipse workbench might contain several views or several editors. Most Eclipse users get along fine without giving this "several views" business a second thought (or even a first thought). But if you care about the terminology surrounding tabs and active views, here's the scoop:

- **Tab:** Something that's impossible to describe except by calling it a "tab." That which we call a tab by any other name would move us as well from one view to another or from one editor to another. The important thing is, views can be *stacked* on top of one another. Eclipse displays stacked views as if they're pages in a tabbed notebook. For example, Figure 3-14 displays one area of the Eclipse workbench. The area contains five views (the Problems view, the Javadoc view, the Declaration view, the Search view, and the Console view). Each view has its own tab.

  A bunch of stacked views is called a *tab group*. To bring a view in the stack to the forefront, you click that view's tab.

  And, by the way, all this stuff about tabs and views holds true for tabs and editors. The only interesting thing is the way Eclipse uses the word "editor." In Eclipse, each tabbed page of the editor area is an individual editor. For example, the Editor area in Figure 3-16 contains three editors (not three tabs belonging to a single editor).

- **Active view or active editor:** In a tab group, the view or editor that's in front.

  In Figure 3-16, the `MyFirstJavaClass.java` editor is the active editor. The `Mortgage.java` and `ThingsILike.java` editors are inactive.

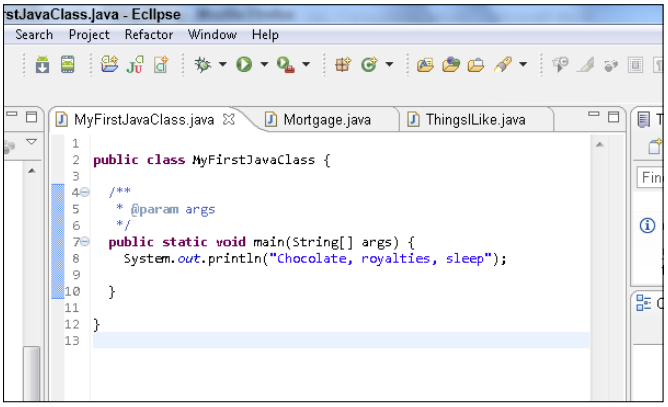


Figure 3-16: The Editor area contains three editors.

What's inside a view or an editor?

The next several terms deal with individual views, individual editors, and individual areas.

- **Toolbar:** The bar of buttons (and other little things) at the top of a view (see Figure 3-17).
- **Menu button:** A downward-pointing arrow in the toolbar. When you click the menu button, a drop-down list of actions appears (see Figure 3-18). Which actions you see in the list varies from one view to another.
- **Close button:** A button that gets rid of a particular view or editor (see Figure 3-19).
- **Chevron:** A double arrow indicating that other tabs should appear in a particular area (but that the area isn't wide enough). The chevron in Figure 3-20 has a little number 2 beside it. The 2 tells you that, in addition to the two visible tabs, two tabs are invisible. Clicking the chevron brings up a hover tip containing the labels of all the tabs (see Figure 3-20).
- **Marker bar:** The vertical ruler on the left edge of the editor area. Eclipse displays tiny alert icons, called *markers*, inside the marker bar. (For example, see Figure 3-12.)

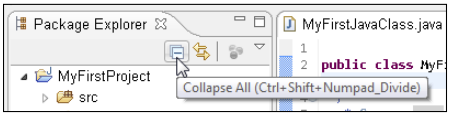


Figure 3-17: The Package Explorer view's toolbar.

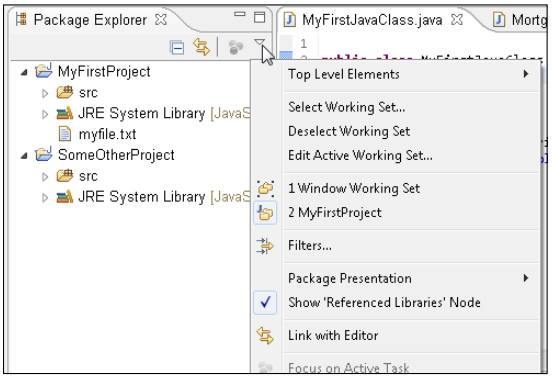


Figure 3-18: Clicking the Package Explorer view's menu button.

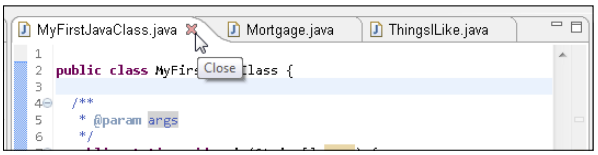


Figure 3-19: An editor's close button.

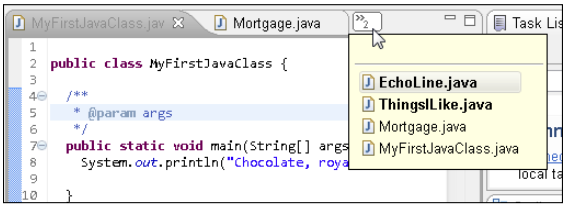


Figure 3-20: The chevron indicates that two editors are hidden.

Returning to the big picture

The next two terms deal with Eclipse's overall look and feel.

- **Layout:** An arrangement of certain views. The layout in Figure 3-3 has seven views, of which four are easily visible:
  - At the far left, you see the Package Explorer view.
  - On the far right, you have the Task List view and the Outline view.
  - Near the bottom, you get the Problems, Javadoc, Declaration, and Console views.

  Along with all these views, the layout contains a single *editor area*. Any and all open editors appear inside this editor area.

- **Perspective:** A very useful layout. If a particular layout is really useful, someone gives that layout a name. And if a layout has a name, you can use the layout whenever you want. For example, the workbench of Figure 3-3 displays Eclipse's *Java perspective*. By default, the Java perspective contains six views in an arrangement very much like the arrangement shown in Figure 3-3.

  The Console view appears in Figure 3-3, but the Console view doesn't always appear as part of the Java perspective. Normally, the Console view appears automatically when you run a text-based Java program. If you want to force the Console view to appear, choose Window➪Show View➪Other. In the resulting Show View dialog box, expand the General branch. Finally, within that General branch, double-click the Console item.

Along with all these views, the Java perspective contains an editor area. (Sure, the editor area has several tabs, but the number of tabs has nothing to do with the Java perspective.)

You can switch among perspectives by choosing Window➪Open Perspective in Eclipse's main menu bar. This book focuses almost exclusively on Eclipse's Java perspective. But if you like poking around, visit some of the other perspectives to get a glimpse of Eclipse's power and versatility.

Part II
Writing Your Own Java Programs

The 5th Wave By Rich Tennant

"Well, isn't this festive – a miniature intranet amidst a swirl of Java applets."

In this part . . .

This part features some of the world's simplest programs. And, as simple as they are, these programs illustrate the fundamental ideas behind all computer code. The ideas include things such as variables, values, types, statements, methods, and lots of other important stuff. This part of the book is your springboard, your launch pad, your virtual catapult.

Chapter 4

Exploring the Parts of a Program

In This Chapter

- Identifying the words in a Java program
- Using punctuation and indentation
- Understanding Java statements and methods

I work in the science building at a liberal arts college. When I walk past the biology lab, I always say a word of thanks under my breath. I'm thankful for not having to dissect small animals. In my line of work, I dissect computer programs instead. Computer programs smell much better than preserved dead animals. Besides, when I dissect a program, I'm not reminded of my own mortality.

In this chapter, I invite you to dissect a program with me. I have a small program, named `ThingsILike`. I cut apart the program and carefully investigate the program's innards. Get your scalpel ready. Here we go!

Checking Out Java Code for the First Time

I have a confession to make. The first time I look at somebody else's computer program, I feel a bit queasy. The realization that I don't understand something (or many things) in the code makes me nervous. I've written hundreds (maybe thousands) of programs, but I still feel insecure when I start reading someone else's code.

The truth is, learning about a computer program is a bootstrapping experience. First I gawk in awe of the program. Then I run the program to see what it does. Then I stare at the program for a while or read someone's explanation of the program and its parts. Then I gawk a little more and run the program again. Eventually, I come to terms with the program. Don't believe the wise guys who say they never go through these steps. Even the experienced programmers approach a new project slowly and carefully.

Behold! A program!

In Listing 4-1, you get a blast of Java code. Like all novice programmers, you're expected to gawk humbly at the code. But *don't be intimidated*. When you get the hang of it, programming is pretty easy. Yes, it's fun, too.

Listing 4-1: A Simple Java Program

```
/*
 * A program to list the good things in life
 * Author: Barry Burd, BeginProg3@allmycode.com
 * February 13, 2012
 */

class ThingsILike {

  public static void main(String args[]) {
    System.out.println("Chocolate, royalties, sleep");
  }
}
```

When I run the program in Listing 4-1, I get the result shown in Figure 4-1: The computer displays the words `Chocolate, royalties, sleep` on the screen. Now I admit that writing and running a Java program is a lot of work just to get the words `Chocolate, royalties, sleep` to appear on somebody's computer screen, but every endeavor has to start somewhere.

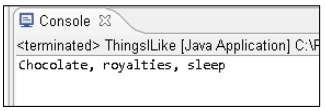


Figure 4-1: Running the program in Listing 4-1.

You can run the code in Listing 4-1 on your computer. Here's how:

1. **Follow the instructions in Chapter 2 for installing Eclipse.**
2. **Next, follow the instructions in the first half of Chapter 3.**

   Those instructions tell you how to run the project named 03-01, which comes in a download from this book's website (`http://allmycode.com/BeginProg3`). To run the code in Listing 4-1, do the same with the 04-01 project, which comes in the same download.

What the program's lines say

If the program in Listing 4-1 ever becomes famous, someone will write a *Cliffs Notes* book to summarize the program. The book will be really short because you can summarize the action of Listing 4-1 in just one sentence. Here's the sentence:

```
Display Chocolate, royalties, sleep
on the computer screen.
```

Now compare the preceding sentence with the bulk in Listing 4-1. Because Listing 4-1 has so many more lines, you may guess that Listing 4-1 has lots of boilerplate code. Well, your guess is correct. You can't write a Java program without writing the boilerplate stuff, but, fortunately, the boilerplate text doesn't change much from one Java program to another. Here's my best effort at summarizing all the Listing 4-1 text in 57 words or fewer:

```
This program lists the good things in life.
Barry Burd wrote this program on February 13, 2012.
Barry realizes that you may have questions about this
code, so you can reach him at BeginProg3@allmycode.com.

This code defines a Java class named ThingsILike.
    Here's the main starting point for the instructions:
        Display Chocolate, royalties, sleep
        on the screen.
```

The rest of this chapter (about 4,500 more words) explains the Listing 4-1 code in more detail.

The Elements in a Java Program

That both English and Java are called *languages* is no coincidence. You use a language to express ideas. English expresses ideas to people, and Java expresses ideas to computers. What's more, both English and Java have things like words, names, and punctuation. In fact, the biggest difference between the two languages is that Java is easier to learn than English. (If English were easy, then computers would understand English. Unfortunately, they can't.)

Take an ordinary English sentence and compare it with the code in Listing 4-1. Here's the sentence:

> Suzanne says "eh" because, as you know, she lives in Canada.

In your high school grammar class, you worried about verbs, adjectives, and other such things. But in this book, you'll think in terms of keywords and identifiers, as summarized in Figure 4-2.

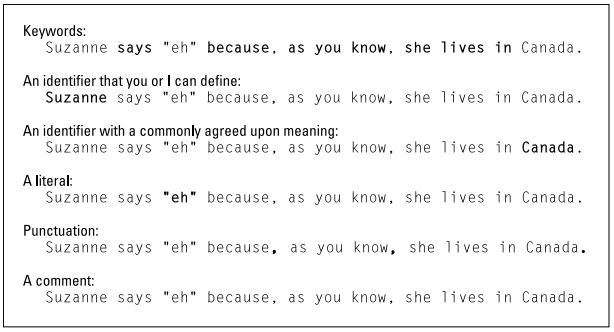

Keywords:
Suzanne says "eh" because, as you know, she lives in Canada.
An identifier that you or I can define:
Suzanne says "eh" because, as you know, she lives in Canada.
An identifier with a commonly agreed upon meaning:
Suzanne says "eh" because, as you know, she lives in Canada.
A literal:
Suzanne says "eh" because, as you know, she lives in Canada.
Punctuation:
Suzanne says "eh" because, as you know, she lives in Canada.
A comment:
Suzanne says "eh" because, as you know, she lives in Canada.

Figure 4-2: The things you find in a simple sentence.

Suzanne's sentence has all kinds of things in it. They're the same kinds of things that you find in a computer program. So here's the plan: Compare the elements in Figure 4-1 with similar elements in Listing 4-1. You already understand English, so you use this understanding to figure out some new things about Java.

But first, here's a friendly reminder: In the next several paragraphs, I draw comparisons between English and Java. As you read these paragraphs, it's important to keep an open mind. For example, in comparing Java with English, I may write that "names of things aren't the same as dictionary words." Sure, you can argue that some dictionaries list proper nouns and that some people have first names like Hope, Prudence, and Spike, but please don't. You'll get more out of the reading if you avoid nitpicking. Okay? Are we still friends?

Keywords

A *keyword* is a dictionary word — a word that's built right into a language.

In Figure 4-2, a word like "says" is a keyword because "says" plays the same role whenever it's used in an English sentence. The other keywords in the Suzanne sentence are "because," "as," "you," "know," "she," "lives," and "in."

Computer programs have keywords, too. In fact, the program in Listing 4-1 uses four of Java's keywords (shown in bold):

```
class ThingsILike {

    public static void main(String args[]) {
```

Each Java keyword has a specific meaning — a meaning that remains unchanged from one program to another. For example, whenever I write a Java program, the word `public` always signals a part of the program that's accessible to any other piece of code.

The java proGRAMMing lanGUage is *case-sensitive*. ThIS MEans that if you change a lowerCASE LETTer in a wORD TO AN UPPercase letter, you chANge the wORD'S MEaning. ChangiNG CASE CAN MakE the enTIRE WORD GO FROM BeiNG MEANINGFul to bEING MEaningless. In Listing 4-1, you can't replace *public* with *Public.* If you do, the WHOLE PROGRAM STOPS WORKING.

This chapter has little or no detail about the meanings of the keywords `class`, `public`, `static`, and `void`. You can peek ahead at the material in other chapters, but you can also get along by cheating. When you write a program, just start with

```
class SomethingOrOther {
```

and then paste the text

```
public static void main(String args[]) {
```

into your code. In your first few programs, this strategy serves you well.

Table 4-1 has a complete list of Java keywords.

Table 4-1 Java Keywords

abstract	continue	for	new
switch	assert	default	goto
package	synchronized	boolean	do
if	private	this	break
double	implements	protected	throw
byte	else	import	public
throws	case	enum	instanceof
return	transient	catch	extends
int	short	try	char
final	interface	static	void
class	finally	long	strictfp
volatile	const	float	native
super	while		

In Java, the words `true`, `false`, and `null` have specific meanings. Like the keywords in Table 4-1, you can't use `true`, `false`, and `null` to mean anything other than what they normally mean in a Java program. But for reasons that concern only the fussiest Java experts, `true`, `false`, and `null` are not called Java keywords. One way or another, if you scribble the words `true`, `false`, and `null` into Table 4-1, you'll be okay.

Here's one thing to remember about keywords: In Java, each keyword has an official, predetermined meaning. The people at Oracle, who have the final say on what constitutes a Java program, have created all of Java's keywords. You can't make up your own meaning for any of the Java keywords. For example, you can't use the word `public` in a calculation:

```
//This is BAD, BAD CODE:
public = 6;
```

If you try to use a keyword this way, then the compiler displays an error message and refuses to translate your source code. It works the same way in English. Have a baby and name it "Because."

> "Let's have a special round of applause for tonight's master of ceremonies — Because O. Borel."

You can do it, but the kid will never lead a normal life.

Despite my ardent claims in this section, two of Java's keywords have no meaning in a Java program. Those keywords — `const` and `goto` — are reserved for non-use in Java. If you try to create a variable named `goto`, Eclipse displays an `Invalid VariableDeclaratorId` error message. The creators of Java figure that, if you use either of the words `const` or `goto` in your code, you should be told politely to move to the C++ programmers' table.

Identifiers that you or I can define

I like the name Suzanne, but if you don't like traditional names, then make up a brand new name. You're having a new baby. Call her "Deneen" or "Chrisanta." Name him "Belton" or "Merk."

A *name* is a word that identifies something, so I'll stop calling these things names and start calling them *identifiers.* In computer programming, an *identifier* is a noun of some kind. An identifier refers to a value, a part of a program, a certain kind structure, or any number of things.

Listing 4-1 has two identifiers that you or I can define on our own. They're the made-up words `ThingsILike` and `args`.

```
class ThingsILike {

    public static void main(String args[]) {
```

Just as the names Suzanne and Chrisanta have no special meaning in English, so the names `ThingsILike` and `args` have no special meaning in Java. In Listing 4-1, I use `ThingsILike` for the name of my program, but I could also have used a name like `GooseGrease`, `Enzyme`, or `Kalamazoo`. I have to put

`(String someName[])` in my program, but I could use `(String args[])`, `(String commandLineArguments[])`, or `(String cheese[])`.

Do as I say, not as I do. Make up sensible, informative names for the things in your Java programs. Names like `GooseGrease` are cute, but they don't help you keep track of your program-writing strategy.

When I name my Java program, I can use `ThinksILike` or `GooseGrease`, but I can't use the word `public`. Words like `class`, `public`, `static`, and `void` are keywords in Java.

The `args` in `(String args[])` holds anything extra that you type when you issue the command to run a Java program. For example, if you get the program to run by typing `java ThingsILike won too 3`, then `args` stores the extra values `won`, `too`, and `3`. As a beginning programmer, you don't need to think about this feature of Java. Just paste `(String args[])` into each of your programs.

Identifiers with agreed-upon meanings

Many people are named Suzanne, but only one country is named Canada. That's because there's a standard, well-known meaning for the word "Canada." It's the country with a red maple leaf on its flag. If you start your own country, you should avoid naming it Canada because naming it Canada would just confuse everyone. (I know, a town in Kentucky is named Canada, but that doesn't count. Remember, you should ignore exceptions like this.)

Most programming languages have identifiers with agreed-upon meanings. In Java, almost all these identifiers are defined in the Java API. Listing 4-1 has five such identifiers. They're the words `main`, `String`, `System`, `out`, and `println`:

```
public static void main(String args[]) {
    System.out.println("Chocolate, royalties, sleep");
}
```

Here's a quick rundown on the meaning of each of these names (and more detailed descriptions appear throughout this book):

- **main:** The main starting point for execution in every Java program.
- **String:** A bunch of text; a row of characters, one after another.
- **System:** A canned program in the Java API. (This program accesses some features of your computer that are outside the direct control of the Java virtual machine (JVM).)
- **out:** The place where a text-based program displays its text. (For a program running in Eclipse, the word `out` represents the Console view.

> To read more about text-based programs, check the first several paragraphs of Chapter 3.)

- **println:** Display text on your computer screen.

Strictly speaking, the meanings of the identifiers in the Java API aren't cast in stone. Although you can make up your own meanings for the words like `System` or `println`, this isn't a good idea. If you did, you would confuse the dickens out of other programmers, who are used to the standard API meanings for these familiar identifier names.

Literals

A *literal* is a chunk of text that looks like whatever value it represents. In Suzanne's sentence (refer to Figure 4-2), "eh" is a literal because "eh" refers to the word "eh."

Programming languages have literals, too. For example, in Listing 4-1, the stuff in quotes is a literal:

```
System.out.println("Chocolate, royalties, sleep");
```

When you run the `ThingsILike` program, you see the words `Chocolate, royalties, sleep` on the screen. In Listing 4-1, the text `"Chocolate, royalties, sleep"` refers to these words, exactly as they appear on the screen (minus the quotation marks).

Most of the numbers that you use in computer programs are literals. If you put the statement

```
mySalary = 1000000.00;
```

in a computer program, then `1000000.00` is a literal. It stands for the number 1000000.00 (one million).

If you don't enjoy counting digits, you can put the following statement in your Java 7 program:

```
mySalary = 1_000_000.00;
```

Starting with Java 7, numbers with underscores are permissible as literals.

In versions of Java before Java 7, you cannot use numbers such as `1_000_000.00` in your code.

Different countries use different number separators and different number formats. For example, in the United States, you write 1,234,567,890,55. In France, you write 1234567890,55. In India, you group digits in sets of two and three. You write 1,23,45,67,890.55. You can't put a statement like `mySalary = 1,000,000.00` in your Java program. Java's numeric literals don't have any commas in them. But you can write `mySalary = 10_00_000.00` for easy-to-read programming in India. And for a program's output, you can display numbers like 1234567890,55 using Java's `Locale` and `NumberFormat` classes.

Punctuation

A typical computer program has lots of punctuation. For example, consider the program in Listing 4-1:

```
class ThingsILike {

  public static void main(String args[]) {
    System.out.println("Chocolate, royalties, sleep");
  }
}
```

Each bracket, each brace, each squiggle of any kind plays a role in making the program meaningful.

In English, you write all the way across one line and then you wrap your text to the start of the next line. In programming, you seldom work this way. Instead, the code's punctuation guides the indenting of certain lines. The indentation shows which parts of the program are subordinate to which other parts. It's as if, in English, you wrote Suzanne's sentence like this:

Suzanne says "eh" because
,
as you know
,
she lives in Canada.

The diagrams in Figures 4-3 and 4-4 show you how parts of the `ThingsILike` program are contained inside other parts. Notice how a pair of curly braces acts like a box. To make the program's structure visible at a glance, you indent all the stuff inside of each box.

I can't emphasize this point enough. If you don't indent your code or if you indent but you don't do it carefully, then your code still compiles and runs correctly. But this successful run gives you a false sense of confidence. The minute you try to update some poorly indented code, you become hopelessly

confused. So take my advice: Keep your code carefully indented at every step in the process. Make your indentation precise, whether you're scratching out a quick test program or writing code for a billionaire customer.

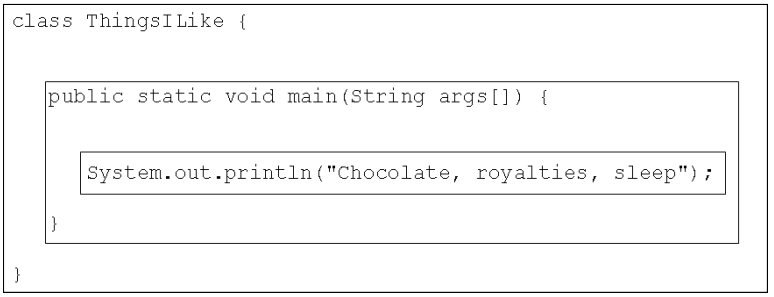


Figure 4-3: A pair of curly braces acts like a box.

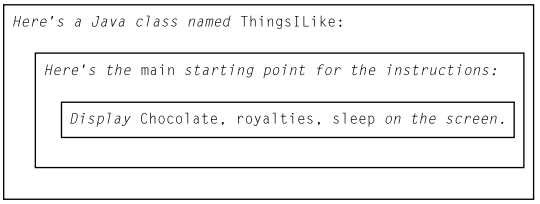

Here's a Java class named ThingsILike:
Here's the main starting point for the instructions:
Display Chocolate, royalties, sleep on the screen.

Figure 4-4: The ideas in a computer program are nested inside of one another.

Eclipse can indent your code automatically for you. Select the `.java` file whose code you want to indent. Then, in Eclipse's main menu, choose Source➪Format. Eclipse rearranges your lines in the editor, indenting things that should be indented and generally making your code look good. (This might not work if your code contains compile-time errors. But it's always worth a try.)

Comments

A *comment* is text that's outside the normal flow. In Figure 4-2, the words "A comment:" aren't part of the Suzanne sentence. Instead, these words are *about* the Suzanne sentence.

The same is true of comments in computer programs. The first five lines in Listing 4-1 form one big comment. The computer doesn't act on this comment. There are no instructions for the computer to perform inside this comment. Instead, the comment tells other programmers something about your code.

Comments are for your own benefit, too. Imagine that you set aside your code for a while and work on something else. When you return later to work on the code again, the comments help you remember what you were doing.

The Java programming language has three different kinds of comments:

- **Traditional comments:** The comment in Listing 4-1 is a *traditional* comment. The comment begins with `/*` and ends with `*/`. Everything between the opening `/*` and the closing `*/` is for human eyes only. Nothing between `/*` and `*/` gets translated by the compiler.

  The second, third, and fourth lines in Listing 4-1 have extra asterisks. I call them "extra" because these asterisks aren't required when you create a comment. They just make the comment look pretty. I include them in Listing 4-1 because, for some reason that I don't entirely understand, most Java programmers add these extra asterisks.

- **End-of-line comments:** Here's some code with end-of-line comments:

```
class ThingsILike {          //Two things are missing

  public static void main(String args[]) {
    System.out.println("sleep"); // Missing from here
  }
}
```

  An *end-of-line* comment starts with two slashes and goes to the end of a line of type.

  You may hear programmers talk about "commenting out" certain parts of their code. When you're writing a program, and something's not working correctly, it often helps to try removing some of the code. If nothing else, you find out what happens when that suspicious code is removed. Of course, you may not like what happens when the code is removed, so you don't want to delete the code completely. Instead, you turn your ordinary Java statements into comments. For example, turn `System.out.println("Sleep");` into `/* System.out.println("Sleep"); */`. This keeps the Java compiler from seeing the code while you try to figure out what's wrong with your program.

- **Javadoc comments:** A special *Javadoc* comment is any traditional comment that begins with an extra asterisk.

```
/**
 * Print a String and then terminate the line.
 */
```

  This is a cool Java feature. The software that you can download from `java.sun.com` includes a little program called `javadoc`. The `javadoc` program looks for these special comments in your code. The program uses these comments to create a brand new web page — a customized documentation page for your code. To find out more about turning Javadoc comments into web pages, visit this book's website (`http://allmycode.com/BeginProg3`).

Understanding a Simple Java Program

The following sections present, explain, analyze, dissect, and otherwise demystify the Java program in Listing 4-1.

What is a method?

You're working as an auto mechanic in an upscale garage. Your boss, who's always in a hurry and has a habit of running words together, says, "FixTheAlternator on that junkyOldFord." Mentally, you run through a list of tasks. "Drive the car into the bay, lift the hood, get a wrench, loosen the alternator belt," and so on. Three things are going on here:

- **You have a name for the thing you're supposed to do.** The name is FixTheAlternator.
- **In your mind, you have a list of tasks associated with the name FixTheAlternator.** The list includes "Drive the car into the bay, lift the hood, get a wrench, loosen the alternator belt," and so on.
- **You have a grumpy boss who's telling you to do all this work.** Your boss gets you working by saying, "FixTheAlternator." In other words, your boss gets you working by saying the name of the thing you're supposed to do.

In this scenario, using the word *method* wouldn't be a big stretch. You have a method for doing something with an alternator. Your boss calls that method into action, and you respond by doing all the things in the list of instructions that you've associated with the method.

Java methods

If you believe all that stuff in the preceding section, then you're ready to read about Java methods. In Java, a *method* is a list of things to do. Every method has a name, and you tell the computer to do the things in the list by using the method's name in your program.

I've never written a program to get a robot to fix an alternator. But, if I did, the program may include a method named `FixTheAlternator`. The list of instructions in my `FixTheAlternator` method would look something like the text in Listing 4-2.

Listing 4-2: A Method Declaration

```
void FixTheAlternator() {
    DriveInto(car, bay);
    Lift(hood);
    Get(wrench);
    Loosen(alternatorBelt);
    ...
}
```

Somewhere else in my Java code (somewhere outside of Listing 4-2), I need an instruction to call my `FixTheAlternator` method into action. The instruction to call the `FixTheAlternator` method into action may look like the line in Listing 4-3.

Listing 4-3: Calling a Method

```
FixTheAlternator(junkyOldFord);
```

Don't scrutinize Listings 4-2 and 4-3 too carefully. All the code in Listings 4-2 and 4-3 is fake! I made up this code so that it looks a lot like real Java code, but it's not real. What's more important, the code in Listings 4-2 and 4-3 isn't meant to illustrate all the rules about Java. So if you have a grain of salt handy, take it with Listings 4-2 and 4-3.

Almost every computer programming language has something akin to Java's methods. If you've worked with other languages, you may remember things like subprograms, procedures, functions, subroutines, `Sub` procedures, or `PERFORM` statements. Whatever you call it in your favorite programming language, a *method* is a bunch of instructions collected together and given a new name.

The declaration, the header, and the call

If you have a basic understanding of what a method is and how it works (see preceding section), you can dig a little deeper into some useful terminology:

- If I'm being lazy, I refer to the code in Listing 4-2 as a *method*. If I'm not being lazy, I refer to this code as a *method declaration*.
- The method declaration in Listing 4-2 has two parts. The first line (the part with the name `FixTheAlternator` in it, up to but not including the open curly brace) is called a *method header*. The rest of Listing 4-2 (the part surrounded by curly braces) is a *method body*.
- The term *method declaration* distinguishes the list of instructions in Listing 4-2 from the instruction in Listing 4-3, which is known as a *method call*.

For a handy illustration of all the method terminology, see Figure 4-5.

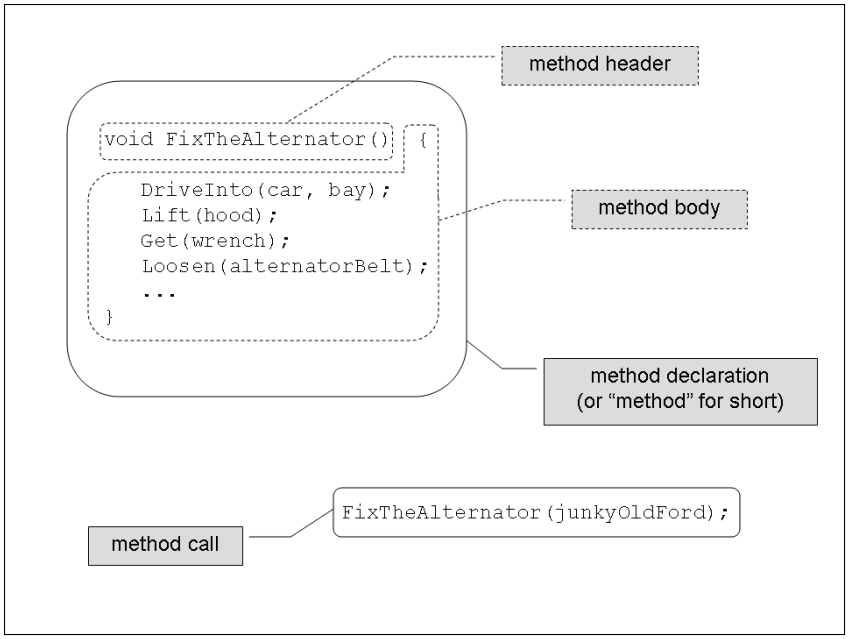


Figure 4-5: The terminology describing methods.

A method's header and body are like an entry in a dictionary. An entry doesn't really use the word that it defines. Instead, an entry tells you what happens if and when you use the word.

> **chocolate** (choc-o-late) *n.* **1.** The most habit-forming substance on earth. **2.** Something you pay for with money from royalties. **3.** The most important nutritional element in a person's diet.
>
> **FixTheAlternator**() Drive the car into the bay, lift the hood, get the wrench, loosen the alternator belt, and then eat some chocolate.

In contrast, a method call is like the use of a word in a sentence. A method call sets some code in motion.

> "I want some chocolate, or I'll throw a fit."
>
> "FixTheAlternator on that junkyOldFord."

A *method's declaration* tells the computer what will happen if you call the method into action. A *method call* (a separate piece of code) tells the computer to actually call the method into action. A method's declaration and the method's call tend to be in different parts of the Java program.

The main method in a program

In Listing 4-1, the bulk of the code is the declaration of a method named *main*. (Just look for the word `main` in the code's method header.) For now, don't worry about the other words in the method header — the words `public`, `static`, `void`, `String`, and `args`. I explain these words (on a need-to-know basis) in the next several chapters.

Like any Java method, the `main` method is a recipe:

```
How to make biscuits:
    Preheat the oven.
    Roll the dough.
    Bake the rolled dough.
```

or

```
How to follow the main instructions in
the ThingsILike code:
    Display Chocolate, royalties, sleep on the screen.
```

The word `main` plays a special role in Java. In particular, you never write code that explicitly calls a `main` method into action. The word `main` is the name of the method that is called into action automatically when the program begins running.

When the `ThingsILike` program runs, the computer automatically finds the program's `main` method and executes any instructions inside the method's body. In the `ThingsILike` program, the `main` method's body has only one instruction. That instruction tells the computer to print `Chocolate, royalties, sleep` on the screen.

None of the instructions in a method are executed until the method is called into action. But if you give a method the name `main`, then that method is called into action automatically.

How you finally tell the computer to do something

Buried deep in the heart of Listing 4-1 is the single line that actually issues a direct instruction to the computer. The line

```
System.out.println("Chocolate, royalties, sleep");
```

tells the computer to display the words `Chocolate, royalties, sleep`. (If you use Eclipse, the computer displays `Chocolate, royalties, sleep` in the Console view.) I can describe this line of code in at least two different ways:

- **It's a statement:** In Java, a direct instruction that tells the computer to do something is called a *statement*. The statement in Listing 4-1 tells the computer to display some text. The statements in other programs may tell the computer to put 7 in a certain memory location or make a window appear on the screen. The statements in computer programs do all kinds of things.
- **It's a method call:** In the "What is a method?" section, earlier in this chapter, I describe something named a "method call." The statement

  ```
  FixTheAlternator(junkyOldFord);
  ```

  is an example of a method call, and so is

  ```
  System.out.println("Chocolate, royalties, sleep");
  ```

  Java has many different kinds of statements. A method call is just one kind.

Ending a statement with a semicolon

In Java, each statement ends with a semicolon. The code in Listing 4-1 has only one statement in it, so only one line in Listing 4-1 ends with a semicolon.

Take any other line in Listing 4-1, like the method header, for example. The method header (the line with the word `main` in it) doesn't directly tell the computer to do anything. Instead, the method header describes some action for future reference. The header announces "Just in case someone ever calls the `main` method, the next few lines of code tell you what to do in response to that call."

Every complete Java statement ends with a semicolon. A method call is a statement, so it ends with a semicolon, but neither a method header nor a method declaration is a statement.

The method named System.out.println

The statement in the middle of Listing 4-1 calls a method named `System.out.println`. This method is defined in the Java API. Whenever you call the `System.out.println` method, the computer displays text on its screen.

Think about names. Believe it or not, I know two people named Pauline Ott. One of them is a nun; the other is physicist. Of course, there are plenty of Paulines in the English-speaking world, just as there are several things named

println in the Java API. So to distinguish the physicist Pauline Ott from the film critic Pauline Kael, I write the full name "Pauline Ott." And, to distinguish the nun from the physicist, I write "Sister Pauline Ott." In the same way, I write either System.out.println or DriverManager.println. The first (which you use often) writes text on the computer's screen. The second (which you don't use at all in this book) writes to a database log file.

Just as Pauline and Ott are names in their own right, so System, out, and println are names in the Java API. But to use println, you must write the method's full name. You never write println alone. It's always System.out.println or some other combination of API names.

The Java programming language is cAsE-sEnSiTiVe. If you change a lowercase letter to an uppercase letter (or vice versa), you change a word's meaning. You can't replace System.out.println with system.out.Println. If you do, your program won't work.

Methods, methods everywhere

Two methods play roles in the ThingsILike program. Figure 4-6 illustrates the situation, and the next few bullets give you a guided tour:

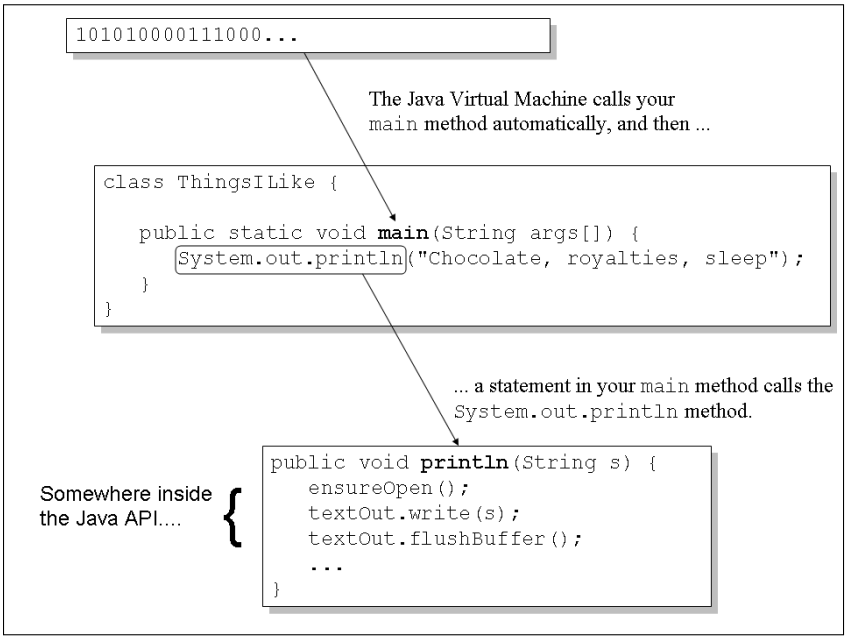


Figure 4-6: Calling the System.out.println method.

- **There's a declaration for a `main` method.** I wrote the `main` method myself. This `main` method is called automatically whenever I start running the `ThingsILike` program.
- **There's a call to the `System.out.println` method.** The method call for the `System.out.println` method is the only statement in the body of the `main` method. In other words, calling the `System.out.println` method is the only thing on the `main` method's to-do list.

  The declaration for the `System.out.println` method is buried inside the official Java API. For a refresher on the Java API, see Chapter 1.

When I say things like "`System.out.println` is buried inside the API," I'm not doing justice to the API. True, you can ignore all the nitty-gritty Java code inside the API. All you need to remember is that `System.out.println` is defined somewhere inside that code. But I'm not being fair when I make the API code sound like something magical. The API is just another bunch of Java code. The statements in the API that tell the computer what it means to carry out a call to `System.out.println` look a lot like the Java code in Listing 4-1.

The Java class

Have you heard the term *object-oriented programming* (also known as *OOP*)? OOP is a way of thinking about computer programming problems — a way that's supported by several different programming languages. OOP started in the 1960s with a language called Simula. It was reinforced in the 1970s with another language named Smalltalk. In the 1980s, OOP took off big time with the language C++.

Some people want to change the acronym, and call it COP — class-oriented programming. That's because object-oriented programming begins with something called a *class*. In Java, everything starts with classes, everything is enclosed in classes, and everything is based on classes. You can't do anything in Java until you've created a class of some kind. It's like being on *Jeopardy*, hearing Alex Trebek say, "Let's go to a commercial" and then interrupting him by saying, "I'm sorry, Alex. You can't issue an instruction without putting your instruction inside a class."

It's important for you to understand what a class really is, so I dare not give a haphazard explanation in this chapter. Instead, I devote much of Chapter 17 to the question, "What is a class?" Anyway, in Java, your `main` method has to be inside a class. I wrote the code in Listing 4-1, so I got to make up a name for my new class. I chose the name `ThingsILike`, so the code in Listing 4-1 starts with the words `class ThingsILike`.

Take another look at Listing 4-1 and notice what happens after the line `class ThingsILike`. The rest of the code is enclosed in curly braces. These braces mark all the stuff inside the class. Without these braces, you'd know where the declaration of the `ThingsILike` class starts, but you wouldn't know where the declaration ends.

It's as if the stuff inside the `ThingsILike` class is in a box. (Refer to Figure 4-3.) To box off a chunk of code, you do two things:

- **You use curly braces:** These curly braces tell the compiler where a chunk of code begins and ends.
- **You indent code:** Indentation tells your human eye (and the eyes of other programmers) where a chunk of code begins and ends.

Don't forget. You have to do both.

Chapter 5

Composing a Program

In This Chapter

- Reading input from the keyboard
- Editing a program
- Shooting at trouble

Just yesterday, I was chatting with my servant, RoboJeeves. (RoboJeeves is an upscale model in the RJ-3000 line of personal robotic life-forms.) Here's how the discussion went:

Me: RoboJeeves, tell me the velocity of an object after it's been falling for 3 seconds in a vacuum.

RoboJeeves: All right, I will. "The velocity of an object after it's been falling for 3 seconds in a vacuum." There, I told it to you.

Me: RoboJeeves, don't give me that smart-alecky answer. I want a number. I want the actual velocity.

RoboJeeves: Okay! "A number; the actual velocity."

Me: RJ, these cheap jokes are beneath your dignity. Can you or can't you tell me the answer to my question?

RoboJeeves: Yes.

Me: "Yes," what?

RoboJeeves: Yes, I either can or can't tell you the answer to your question.

Me: Well, which is it? Can you?

RoboJeeves: Yes, I can.

Me: Then do it. Tell me the answer.

RoboJeeves: The velocity is 153,984,792 miles per hour.

Me: (After pausing to think . . .) RJ, I know you never make a mistake, but that number, 153,984,792, is much too high.

> *RoboJeeves:* Too high? That's impossible. Things fall very quickly on the giant planet Mangorrrrkthongo. Now, if you wanted to know about objects falling on Earth, you should have said so in the first place.

Sometimes that robot rubs me the wrong way. The truth is, RoboJeeves does whatever I tell him to do — nothing more and nothing less. If I say "Feed the cat," then RJ says, "Feed it to whom? Which of your guests will be having cat for dinner?"

Computers Are Stupid

Handy as they are, all computers do the same darn thing. They do *exactly* what you tell them to do, and that's sometimes very unfortunate. For example, in 1962, a Mariner spacecraft to Venus was destroyed just 4 minutes after its launch. Why? It was destroyed because of a missing keystroke in a FORTRAN program. Around the same time, NASA scientists caught an error that could have trashed the Mercury space flights. (Yup! These were flights with people on board!) The error was a line with a period instead of a comma. (A computer programmer wrote `DO 10 I=1.10` instead of `DO 10 I=1,10`.)

With all due respect to my buddy RoboJeeves, he and his computer cousins are all incredibly stupid. Sometimes they look as if they're second-guessing us humans, but actually they're just doing what other humans told them to do. They can toss virtual coins and use elaborate schemes to mimic creative behavior, but they never really think on their own. If you say, "Jump," then they do what they're programmed to do in response to the letters J-u-m-p.

So when you write a computer program, you have to imagine that a genie has granted you three wishes. Don't ask for eternal love because, if you do, then the genie will give you a slobbering, adoring mate — someone that you don't like at all. And don't ask for a million dollars unless you want the genie to turn you into a bank robber.

Everything you write in a computer program has to be very precise. Take a look at an example. . . .

A Program to Echo Keyboard Input

Listing 5-1 contains a small Java program. The program lets you type one line of characters on the keyboard. As soon as you press Enter, the program displays a second line that copies whatever you typed.

Listing 5-1: A Java Program

```
import java.util.Scanner;

class EchoLine {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);

        System.out.println(myScanner.nextLine());
    }
}
```

Figure 5-1 shows a run of the `EchoLine` code (the code in Listing 5-1). The text in the figure is a mixture of my own typing and the computer's responses.

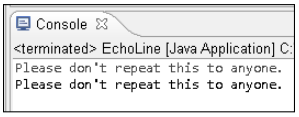


Figure 5-1: What part of the word "don't" do you not understand?

In Figure 5-1, I type the first line (the first `Please don't repeat this to anyone` line) and the computer displays the second line. Here's what happens when you run the code in Listing 5-1:

1. **At first, the computer does nothing.**

   The computer is waiting for you to type something.

2. **You click your mouse inside Eclipse's Console view.**

   As a result you see a cursor on the left edge of Eclipse's Console view, as shown in Figure 5-2.

Figure 5-2: The computer waits for you to type something.

3. **You type one line of text — any text at all (see Figure 5-3).**

Figure 5-3: You type a sentence.

4. **You press Enter, and the computer displays another copy of the line that you typed, as shown in Figure 5-4.**

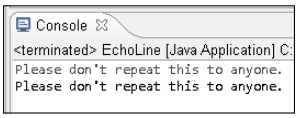


Figure 5-4: The computer echoes your input.

After displaying a copy of your input, the program's run comes to an end.

Typing and running a program

This book's website (`http://allmycode.com/BeginProg3`) has a link for downloading all the book's Java programs. After you download the programs, you can follow instructions in Chapter 2 to add the programs to your Eclipse workspace. Then, to test the code in Listing 5-1, you can run the readymade 05-01 project.

But instead of running the readymade code, I encourage you to start from scratch — to type Listing 5-1 yourself and then to test your newly created code. Just follow these steps:

1. **Launch Eclipse.**
2. **From Eclipse's menu bar, choose File➪New➪Java Project.**

   Eclipse's New Java Project dialog box appears.
3. **In the dialog box's Project Name field, type** MyNewProject**.**
4. **Click Finish.**

Clicking Finish brings you back to the Eclipse workbench, with `MyNewProject` in the Package Explorer. The next step is to create a new Java source code file.

5. **In the Package Explorer, select `MyNewProject` and then, in Eclipse's main menu, choose File➪New➪Class.**

   Eclipse's New Java Class dialog box appears.

6. **In the New Java Class dialog box's Name field, type the name of your new class.**

   In this example, use the name EchoLine. Spell EchoLine exactly the way I spell it in Listing 5-1, with a capital E, a capital L, and no blank space.

   In Java, consistent spelling and capitalization are very important. If you're not consistent within a particular program, then the program will probably have some nasty, annoying compile-time errors.

   Optionally, you can put a check mark in the box labeled `public static void main(String[] args)`. If you leave the box unchecked, you'll have a bit more typing to do when you get to Step 8. Either way (checked or unchecked) it's no big deal.

7. **Click Finish.**

   Clicking Finish brings you back to the Eclipse workbench. An editor in this workbench has a tab named EchoLine.java.

8. **Type the program of Listing 5-1 in the EchoLine.java editor.**

   Copy the code exactly as you see it in Listing 5-1.

   - Spell each word exactly the way I spell it in Listing 5-1.
   - Capitalize each word exactly the way I do in Listing 5-1.
   - Include all the punctuation symbols — the dots, the semicolons, everything.
   - Double-check the spelling of the word `println`. Make sure that each character in the word `println` is a lowercase letter. (In particular, the `l` in `ln` is a letter, not a digit.)

   If you typed everything correctly, you don't see any error markers in the editor.

   If you see error markers, then go back and compare everything you typed with the stuff in Listing 5-1. Compare every letter, every word, every squiggle, every smudge.

9. **Make any changes or corrections to the code in the editor.**

   When at last you see no error markers, you're ready to run the program.

10. **Select the EchoLine class either by clicking inside the editor or by clicking the MyNewProject branch in the Package Explorer.**

11. **In Eclipse's main menu, choose Run➪Run As➪Java Application.**

    Your new Java program runs, but nothing much happens.

12. **Click your mouse inside Eclipse's Console view.**

    As a result, a cursor sits on the left edge of Eclipse's Console view. (Refer to Figure 5-2.) The computer is waiting for you to type something.

    If you forget to click your mouse inside the Console view, Eclipse may not send your keystrokes to the running Java program. Instead, Eclipse may send your keystrokes to the editor or (strangely enough) to the Package Explorer.

13. **Type a line of text and then press Enter.**

    In response, the computer displays a second copy of your line of text. Then the program's run comes to an end. (Refer to Figure 5-4.)

If this list of steps seems a bit sketchy, you can find much more detail in Chapter 3. (Look first at the section in Chapter 3 about compiling and running a program.) For the most part, the steps here in Chapter 5 are a quick summary of the material in Chapter 3. The big difference is that in Chapter 3, I don't encourage you to type the program yourself.

So what's the big deal when you type the program yourself? Well, lots of interesting things can happen when you apply fingers to keyboard. That's why the second half of this chapter is devoted to troubleshooting.

How the EchoLine program works

When you were a tiny newborn, resting comfortably in your mother's arms, she told you how to send characters to the computer screen:

```
System.out.println(whatever text you want displayed);
```

What she didn't tell you was how to fetch characters from the computer keyboard. There are lots of ways to do it, but the one I recommend in this chapter is

```
myScanner.nextLine()
```

Now, here's the fun part. Calling the `nextLine` method doesn't just scoop characters from the keyboard. When the computer runs your program, the computer *substitutes whatever you type on the keyboard* in place of the text `myScanner.nextLine()`.

To understand this, look at the statement in Listing 5-1:

```
System.out.println(myScanner.nextLine());
```

When you run the program, the computer sees your call to `nextLine` and stops dead in its tracks. (Refer to Figure 5-2.) The computer waits for you to type a line of text. So (refer to Figure 5-3) you type this line:

```
Hey, there's an echo in here.
```

The computer substitutes this entire `Hey` line for the `myScanner.nextLine()` call in your program. The process is illustrated in Figure 5-5.

The call to `myScanner.nextLine()` is nestled inside the `System.out.println` call. So when all is said and done, the computer behaves as if the statement in Listing 5-1 looks like this:

```
System.out.println("Hey, there's an echo in here.");
```

The computer displays another copy of the text `Hey, there's an echo in here.` on the screen. That's why you see two copies of the `Hey` line in Figure 5-4.

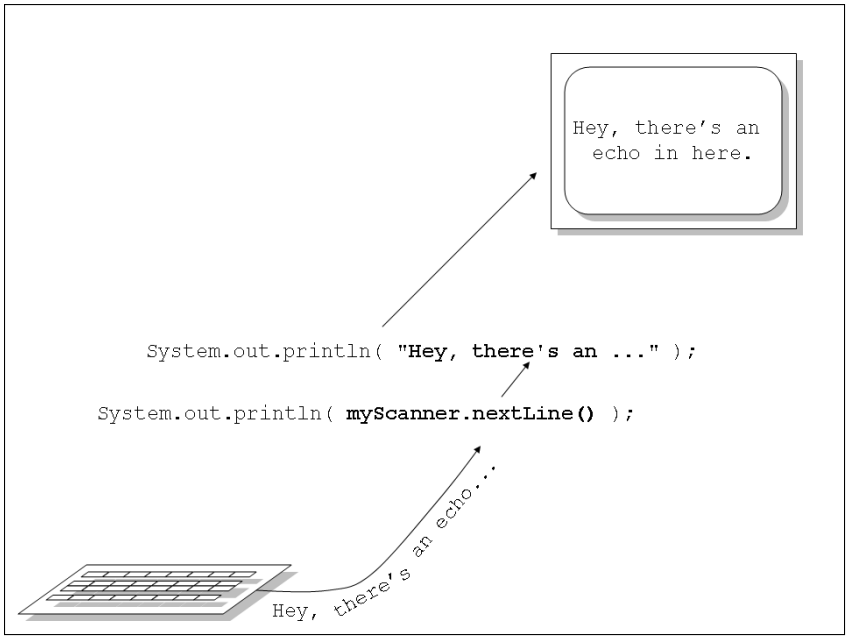


Figure 5-5: The computer substitutes text in place of the `nextLine` call.

Getting numbers, words, and other things

In Listing 5-1, the words `myScanner.nextLine()` get an entire line of text from the computer keyboard. So if you type

```
Testing 1 2 3
```

the program in Listing 5-1 echoes back your entire `Testing 1 2 3` line of text.

Sometimes you don't want a program to get an entire line of text. Instead, you want the program to get a piece of a line. For example, when you type `1 2 3`, you may want the computer to get the number `1`. (Maybe the number `1` stands for one customer or something like that.) In such situations, you don't put `myScanner.nextLine()` in your program. Instead, you use `myScanner.`**`nextInt`**`()`.

Table 5-1 shows you a few variations on the `myScanner.next` business. Unfortunately, the table's entries aren't very predictable. To read a line of input, you call `nextLine`. But to read a word of input, you don't call `nextWord`. (The Java API has no `nextWord` method.) Instead, to read a word, you call `next`.

Table 5-1 Some Scanner Methods

To Read This . . .	*. . . Make This Method Call*
A number with no decimal point in it	`nextInt()`
A number with a decimal point in it	`nextDouble()`
A word (ending in a blank space, for example)	`next()`
A line (or what remains of a line after you've already read some data from the line)	`nextLine()`
A single character (such as a letter, a digit, or a punctuation character)	`findWithinHorizon(".",0).charAt(0)`

Also, the table's story has a surprise ending. To read a single character, you don't call `next`*`Something`*. Instead, you can call the bizarre `findWithinHorizon(".",0).charAt(0)` combination of methods. (You'll have to excuse the folks who created the `Scanner` class. They created `Scanner` from a specialized point of view.)

A quick look at the Scanner

In this chapter, I advise you to ignore any meanings behind the lines `import java.util.Scanner` and `Scanner myScanner`, *etc*. Just paste these two lines mindlessly in your code and then move on.

Of course, you may not want to take my advice. You may not like ignoring things in your code. If you happen to be such a stubborn person, I have a few quick facts for you.

- **The word `Scanner` is defined in the Java API.**

  A `Scanner` is something you can use for getting input.

  This `Scanner` class belongs to Java versions 5.0 and higher. If you use version Java 1.4.2, then you don't have access to the `Scanner` class. (You see an error marker when you type Listing 5-1.)

- **The words `System` and `in` are defined in the Java API.**

  Taken together, the words `System.in` stand for the computer keyboard.

  In later chapters, you see things like `new Scanner(new File("myData.txt"))`. In those chapters, I replace `System.in` with the words `new File("myData.txt")` because I'm not getting input from the keyboard. Instead, I'm getting input from a file on the computer's hard drive.

- **The word `myScanner` doesn't come from the Java API.**

  The word `myScanner` is a Barry Burd creation. Instead of `myScanner`, you can use `readingThingie` (or any other name you want to use as long as you use the name consistently). So, if you want to be creative, you can write

```
Scanner readingThingie =
    new Scanner(System.in);

System.out.println
    (readingThingie.nextLine());
```

  The revised Listing 5-1 (with `readingThingie` instead of `myScanner`) compiles and runs without a hitch.

- **The line `import java.util.Scanner` is an example of an *import declaration*.**

  An optional import declaration allows you to abbreviate names in the rest of your program. You can remove the import declaration from Listing 5-1. But if you do, you must use the `Scanner` class's *fully qualified name* throughout your code. Here's how:

```
class EchoLine {

  public static void main
                   (String args[]) {

    java.util.Scanner myScanner
          = new java.util.Scanner
                        (System.in);

    System.out.println
          (myScanner.nextLine());
  }
}
```

To see some of the table's methods in action, check other program listings in this book. Chapters 6, 7, and 8 have some particularly nice examples.

Type two lines of code and don't look back

Buried innocently inside Listing 5-1 are two extra lines of code. These lines help the computer read input from the keyboard. The two lines are

```
import java.util.Scanner;

        Scanner myScanner = new Scanner(System.in);
```

Concerning these two lines, I have bad news and good news.

- **The bad news is, the reasoning behind these lines is difficult to understand.** That's especially true here in Chapter 5, where I introduce Java's most fundamental concepts.
- **The good news is, you don't have to understand the reasoning behind these two lines.** You can copy and paste these lines into any program that gets input from the keyboard. You don't have to change the lines in any way. These lines work without any modifications in all kinds of Java programs.

Just be sure to put these lines in the right places:

- Make the `import java.util.Scanner` line be the first line in your program.
- Put the `Scanner myScanner = new Scanner(System.in)` line inside your `main` method immediately after the `public static void main(String args[]) {` line.

At some point in the future, you may have to be more careful about the positioning of these two lines. But for now, the rules I give will serve you well.

Expecting the Unexpected

Not long ago, I met an instructor with an interesting policy. He said, "Sometimes when I'm lecturing, I compose a program from scratch on the computer. I do it right in front of my students. If the program compiles and runs correctly on the first try, I expect the students to give me a big round of applause."

At first, you may think this guy has an enormous ego, but you have to put things in perspective. It's unusual for a program to compile and run correctly the first time. There's almost always a typo or another error of some kind.

So this section deals with the normal, expected errors that you see when you compile and run a program for the first time. Everyone makes these mistakes, even the most seasoned travelers. The key is keeping a cool head. Here's my general advice:

- **Don't expect a program that you type to compile the first time.**

  Be prepared to return to your editor and fix some mistakes.

- **Don't expect a program that compiles flawlessly to run correctly.**

  Even with no error markers in Eclipse's editor, your program might still contain flaws. After Eclipse compiles your program, you still have to run the program successfully. That is, your program should finish its run and display the correct output.

  You compile and then you run. Getting a program to compile without errors is the easier of the two tasks.

- **Read what's in the Eclipse editor, not what you assume is in the Eclipse editor.**

  Don't assume that you've typed words correctly, that you've capitalized words correctly, or that you've matched curly braces or parentheses correctly. Compare the code you typed with any sample code that you have. Make sure that every detail is in order.

- **Be patient.**

  Every good programming effort takes a long time to get right. If you don't understand something right away, then be persistent. Stick with it (or put it away for a while and come back to it). There's nothing you can't understand if you put in enough time.

- **Don't become frustrated.**

  Don't throw your pie crust. Frustration (not lack of knowledge) is your enemy. If you're frustrated, you can't accomplish anything.

- **Don't think you're the only person who's slow to understand.**

  I'm slow, and I'm proud of it. (Kelly, Chapter 6 will be a week late.)

- **Don't be timid.**

  If your code isn't working and you can't figure out why it's not working, then ask someone. Post a message on `groups.google.com` or send me an e-mail message. (Send it to `BeginProg3@allmycode.com`.) And don't be afraid of anyone's snide or sarcastic answer. (For a list of gestures you can make in response to peoples' snotty answers, see Appendix Z.)

Diagnosing a problem

The "Typing and running a program" section, earlier in this chapter, tells you how to run the `EchoLine` program. If all goes well, your screen ends up looking like the one shown in Figure 5-1. But things don't always go well. Sometimes your finger slips, inserting a typo into your program. Sometimes you ignore one of the details in Listing 5-1, and you get a nasty error message.

Of course, some things in Listing 5-1 are okay to change. Not every word in Listing 5-1 is cast in stone. So here's a nasty wrinkle — I can't tell you that you must always retype Listing 5-1 exactly as it appears. Some changes are okay; others are not. Keep reading for some "f'rinstances."

Case sensitivity

Java is *case-sensitive*. Among other things, case-sensitive means that, in a Java program, the letter `P` isn't the same as the letter `p`. If you send me some fan mail and start with "Dear barry" instead of "Dear Barry," then I still know what you mean. But Java doesn't work that way.

So change just one character in a Java program and instead of an uneventful compilation, you get a big headache! Change `p` to `P` like so:

```
//The following line is incorrect:
System.out.Println(myScanner.nextLine());
```

When you type the program in Eclipse's editor, you get the ugliness shown in Figure 5-6.

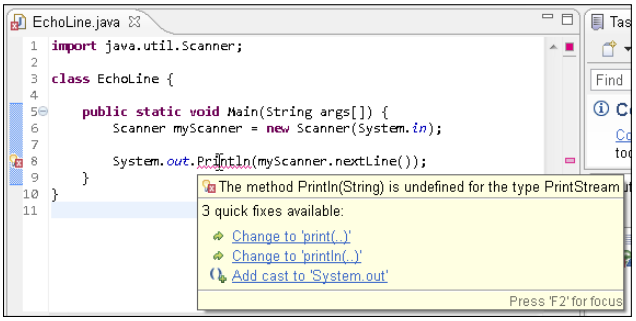


Figure 5-6: The Java compiler understands `println`, but not `Println`.

When you see error markers and quick fixes like the ones in Figure 5-6, your best bet is to stay calm and read the messages carefully. Sometimes, the messages contain useful hints. (Of course, sometimes they don't.) The message in Figure 5-6 is `The method Println(String) is undefined for the type PrintStream`. In plain English, this means "The Java compiler

can't interpret the word `Println`." (The message stops short of saying, "Don't type the word `Println`, you dummy!" In any case, if the computer says you're one of us Dummies, you should take it as a compliment.) Now, there are plenty of reasons why the compiler may not be able to understand a word like `Println`. But, for a beginning programmer, you should check two important things right away:

- **Have you spelled the word correctly?**

  Did you accidentally type `printlin` (with a digit `1`) instead of `println`?

- **Have you capitalized all letters correctly?**

  Did you incorrectly type `Println` or `PrintLn` instead of `println`?

Either of these errors can send the Java compiler into a tailspin. So compare your typing with the approved typing word for word (and letter for letter). When you find a discrepancy, go back to the editor and fix the problem. Then try compiling the program again.

As you type a program in Eclipse's editor, Eclipse tries to compile the program. When Eclipse finds a compile-time error, the editor usually displays at least three red error markers (see Figure 5-6). The marker in the editor's left margin has an X-like marking and sometimes a tiny light bulb. The marker in the right margin is a small rectangle. The marker in the middle is a jagged red underline. If you hover your mouse over any of these markers, Eclipse displays a message that attempts to describe the nature of the error. If you hover over the jagged line, Eclipse displays a message and possibly a list of suggested solutions. (Each suggested solution is a called a *quick fix*.) If you right-click the left margin's marker (or control-click on a Mac) and choose Quick Fix in the resulting context menu, Eclipse displays the suggested solutions. To have Eclipse modify your code automatically (using a suggestion from the quick-fix list), either single-click or double-click the item in the quick-fix list. (That is, single-click anything that looks like a link; double-click anything that doesn't look like a link.)

Omitting punctuation

In English and in, Java using the; proper! punctuation is important)

Take, for example, the semicolons in Listing 5-1. What happens if you forget to type a semicolon?

```
//The following code is incorrect:

    System.out.println(myScanner.nextLine())
}
```

If you leave off the semicolon, you get the message shown in Figure 5-7.

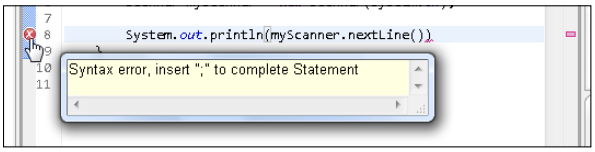


Figure 5-7: A helpful error message.

A message like the one in Figure 5-8 makes your life much simpler. I don't have to explain the message, and you don't have to puzzle over the message's meaning. Just take the message `insert ";" to complete Statement` on its face value. Insert the semicolon between the end of the `System.out.println(myScanner.nextLine())` statement and whatever code comes after the statement. (For code that's easiest to read and understand, tack on the semicolon at the end of the `System.out.println(myScanner.nextLine())` statement.)

Using too much punctuation

In junior high school, my English teacher said I should use a comma whenever I would normally pause for a breath. This advice doesn't work well during allergy season, when my sentences have more commas in them than words. Even as a paid author, I have trouble deciding where the commas should go, so I often add extra commas for good measure. This makes more work for my copy editor, Kelly, who has a trash can full of commas by the desk in her office.

It's the same way in a Java program. You can get carried away with punctuation. Consider, for example, the `main` method header in Listing 5-1. This line is a dangerous curve for novice programmers.

For information on the terms *method header* and *method body,* see Chapter 4.

Normally, you shouldn't be ending a method header with a semicolon. But people add semicolons anyway. (Maybe, in some subtle way, a method header looks like it should end with a semicolon.)

```
//The following line is incorrect:
public static void main(String args[]); {
```

If you add this extraneous semicolon to the code in Listing 5-1, you get the message shown in Figure 5-8.

Why can't the computer fix it?

How often do you get to finish someone else's sentence? "Please," says your supervisor, "go over there and connect the . . ."

"Wires," you say. "I'll connect the wires." If you know what someone means to say, why wait for them to say it?

This same question comes up in connection with computer error messages. Take a look at the message in Figure 5-7. The computer expects a semicolon after the statement on line 8. Well, Mr. Computer, if you know where you want a semicolon, then just add the semicolon and be done with it. Why are you bothering me about it?

The answer is simple. The computer isn't interested in taking any chances. What if you *don't* really want a semicolon after the statement on line 8? What if the missing semicolon represents a more profound problem? If the computer added the extra semicolon, it could potentially do more harm than good.

Returning to you and your supervisor . . .

Boom! A big explosion. "Not the wires, you Dummy. The dots. I wanted you to connect the dots."

"Sorry," you say.

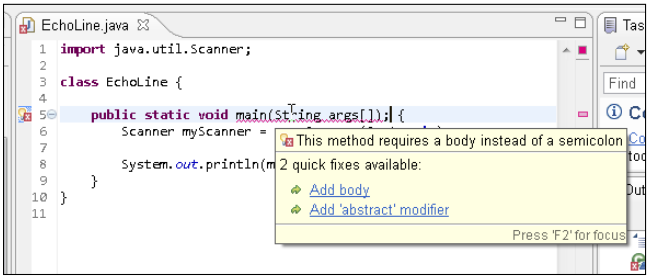


Figure 5-8: An unwanted semicolon messes things up.

The error message and quick fixes in Figure 5-8 are a bit misleading. The message starts with `This method requires a body`. But the method has a body. Doesn't it?

When the computer tries to compile `public static void main(String args[]);` (ending with a semicolon), the computer gets confused. I illustrate the confusion in Figure 5-9. Your eye sees an extra semicolon, but the computer's eye interprets this as a method without a body. So that's the error message — the computer says `This method requires a body instead of a semicolon`.

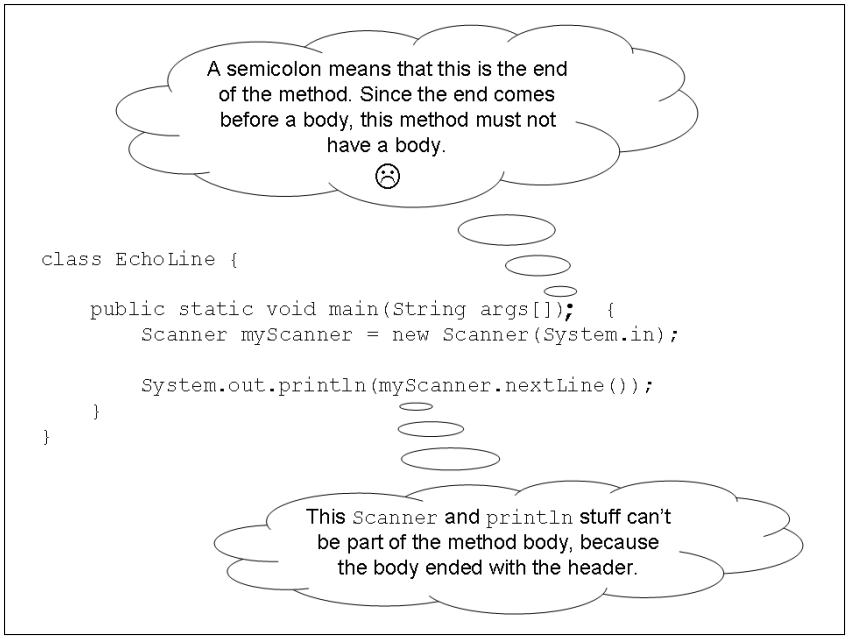

A semicolon means that this is the end of the method. Since the end comes before a body, this method must not have a body.
class EchoLine {
public static void main(String args[]); {
Scanner myScanner = new Scanner(System.in);
System.out.println(myScanner.nextLine());
}
}
This Scanner and println stuff can't be part of the method body, because the body ended with the header.

Figure 5-9: What's on this computer's mind?

If you select the Add Body quick fix, Eclipse creates the following (really horrible) code:

```
import java.util.Scanner;

class EchoLine {

  public static void main(String args[]) {
  }

  {
    Scanner myScanner = new Scanner(System.in);

    System.out.println(myScanner.nextLine());
  }
}
```

This "fixed" code has no compile-time errors. But when you run this code, nothing happens. The program starts running and then stops running with nothing in Eclipse's Console view.

We all know that a computer is a very patient, very sympathetic machine. That's why the computer looks at your code and decides to give you one more chance. The computer remembers that Java has an advanced feature in which you write a method header without writing a method body. When you do this, you get what's called an *abstract method* — something that I don't

use at all in this book. Anyway, in Figure 5-9, the computer sees a header with no body. So the computer says to itself, "I know! Maybe the programmer is trying to write an abstract method. The trouble is, an abstract method's header has to have the word `abstract` in it. I should remind the programmer about that." So the computer offers the `add 'abstract' modifier` quick fix in Figure 5-9.

One way or another, you can't interpret the error message and the quick fixes in Figure 5-9 without reading between the lines. So here are some tips to help you decipher murky messages:

- **Avoid the knee-jerk response.**

  Some people see the `add 'abstract' modifier` quick fix in Figure 5-9 and wonder where they can add a modifier. Unfortunately, this isn't the right approach. If you don't know what an `'abstract' modifier` is, then chances are you didn't mean to use an abstract modifier in the first place.

- **Stare at the bad line of code for a long, long time.**

  If you look carefully at the `public static . . .` line in Figure 5-9, then eventually you'll notice that it's different from the corresponding line in Listing 5-1. The line in Listing 5-1 has no semicolon, but the line in Figure 5-9 has a semicolon.

  Of course, you won't always be starting with some prewritten code like the stuff in Listing 5-1. That's where practice makes perfect. The more code you write, the more sensitive your eyes will become to things like extraneous semicolons and other programming goofs.

Too many curly braces

You're looking for the nearest gas station, so you ask one of the locals. "Go to the first traffic light and make a left," says the local. You go straight for a few streets and see a blinking yellow signal. You turn left at the signal and travel for a mile or so. What? No gas station? Maybe you mistook the blinking signal for a real traffic light.

You come to a fork in the road. "The directions said nothing about a fork. Which way should I go?" You veer right, but a minute later, you're forced onto a highway. You see a sign that says, "Next Exit 24 Miles." Now you're really lost, and the gas gauge points to "S." (The "S" stands for "Stranded.")

So here's what happened: You made an honest mistake. You shouldn't have turned left at the yellow blinking light. That mistake alone wasn't so terrible. But that first mistake lead to more confusion, and eventually, your choices made no sense at all. If you hadn't turned at the blinking light, you'd never have encountered that stinking fork in the road. Then, getting on the highway was sheer catastrophe.

Is there a point to this story? Of course there is. A computer can get itself into the same sort of mess. The computer notices an error in your program. Then, metaphorically speaking, the computer takes a fork in the road — a fork based on the original error — a fork for which none of the alternatives leads to good results.

Here's an example. You're retyping the code in Listing 5-1, and you mistakenly type an extra curly brace:

```
//The following code is incorrect:
import java.util.Scanner;

class EchoLine {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
    }
        System.out.println(myScanner.nextLine());
    }
}
```

In Eclipse's editor, you hover over the leftmost marker. You see the messages shown in Figure 5-10.

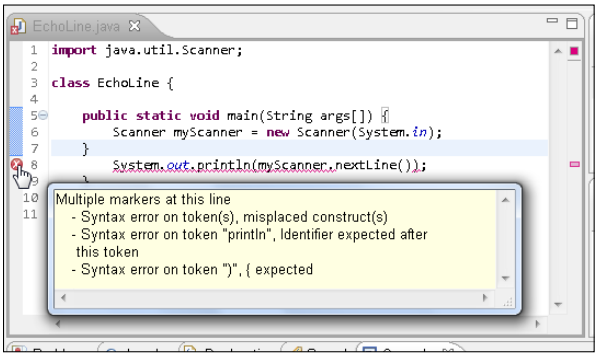

EchoLine.java
1 import java.util.Scanner;
2
3 class EchoLine {
4
5 public static void main(String args[]) {
6 Scanner myScanner = new Scanner(System.in);
7 }
8 System.out.println(myScanner.nextLine());
10
11
Multiple markers at this line
- Syntax error on token(s), misplaced construct(s)
- Syntax error on token "println", Identifier expected after this token
- Syntax error on token ")", { expected

Figure 5-10: Three error messages.

Eclipse is confused because the call to `System.out.println` is completely out of place. Eclipse displays three messages — something about `println`, something about the parenthesis, and something very cryptic concerning misplaced constructs. None of these messages addresses the cause of the problem. Eclipse is trying to make the best of a bad situation, but at this point, you shouldn't believe a word that Eclipse says.

Computers aren't smart animals, and if someone programs the Eclipse to say `misplaced construct(s)`, then that's exactly what the Eclipse says. (Some people say that computers make them feel stupid. For me, it's the opposite. A computer reminds me how dumb a machine can be and how smart a person can be. I like that.)

So when you see a bunch of error messages, read each error message carefully. Ask yourself what you can learn from each message. But don't take each message as the authoritative truth. When you've exhausted your efforts with Eclipse's messages, return to your efforts to stare carefully at the code.

If you get more than one error message, always look carefully at each message in the bunch. Sometimes a very helpful message hides among a bunch of not-so-helpful messages.

Misspelling words (and other missteps)

You've found an old family recipe for deviled eggs (one of my favorites). You follow every step as carefully as you can, but you leave out the salt because of your grandmother's high blood pressure. You hand your grandmother an egg (a finished masterpiece). "Not enough pepper," she says, and she walks away.

The next course is beef bourguignon. You take an unsalted slice to dear old Granny. "Not sweet enough," she groans, and she leaves the room. "But that's impossible," you think. "There's no sugar in beef bourguignon. I left out the salt." Even so, you go back to the kitchen and prepare mashed potatoes. You use unsalted butter, of course. "She'll love it this time," you think.

"Sour potatoes! Yuck!" Granny says, as she goes to the sink to spit it all out. Because you have a strong ego, you're not insulted by your grandmother's behavior. But you're somewhat confused. Why is she saying such different things about three unsalted recipes? Maybe there are some subtle differences that you don't know about.

Well, the same kind of thing happens when you're writing computer programs. You can make the same kind of mistake twice (or at least, make what you think is the same kind of mistake twice) and get different error messages each time.

For example, if you change the spelling or capitalization of `println` in Listing 5-1, Eclipse tells you the method `is undefined for the type PrintStream`. But if you change `System` to `system`, Eclipse says that `system cannot be resolved`. And with `System` misspelled, Eclipse doesn't notice whether `println` is spelled correctly or not.

In Listing 5-1, if you change the spelling of `args`, then nothing goes wrong. The program compiles and runs correctly. But if you change the spelling of `main`, you face some unusual difficulties. (If you don't believe me, read the "Runtime error messages" section.)

Still in Listing 5-1, change the number equal signs in the `Scanner myScanner = new Scanner(System.in)` line. With one equal sign, everybody's happy. If you accidentally type two equal signs (`Scanner myScanner == new Scanner(System.in)`), Eclipse steers you back on course, telling you `Syntax error on token "==", = expected` (see Figure 5-11). But if you go crazy and type four equal signs or if you type no equal signs at all, Eclipse misinterprets everything and suggests that you `insert ";" to complete BlockStatements`. Unfortunately, inserting a semicolon is no help at all (see Figure 5-12).

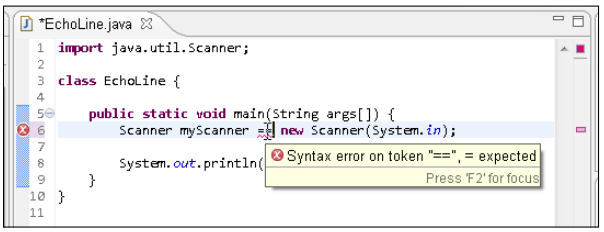


Figure 5-11: Remove the second of two equal signs.

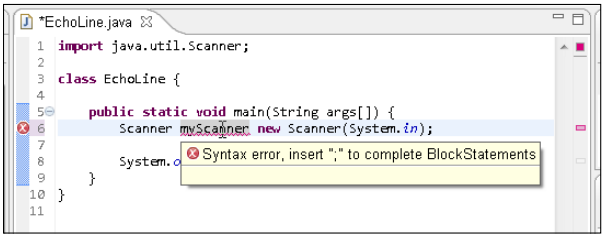


Figure 5-12: You're missing an equal sign, but Eclipse fails to notice.

So remember: Java responds to errors in many different ways. Two changes in your code might look alike, but similar changes don't always lead to similar results. Each problem in your code requires its own individualized attention.

Here's a useful exercise: Start with a working Java program. After successfully running the code, make a change that intentionally introduces errors. Look carefully at each error message and ask yourself whether the message would help you diagnose the problem. This exercise is great because it helps you think of errors as normal occurrences and gives you practice analyzing messages when you're not under pressure to get your program to run correctly.

Runtime error messages

Up to this point in the chapter, I describe errors that crop up when you compile a program. Another category of errors hides until you run the program. A case in point is the improper spelling or capitalization of the word `main`.

Assume that, in a moment of wild abandon, you incorrectly spell `main` with a capital `M`:

```
//The following line is incorrect:
public static void Main(String args[]) {
```

When you type the code, everything is hunky-dory. You don't see any error markers.

But then you try to run your program. At this point, the bits hit the fan. The catastrophe is illustrated in Figure 5-13.

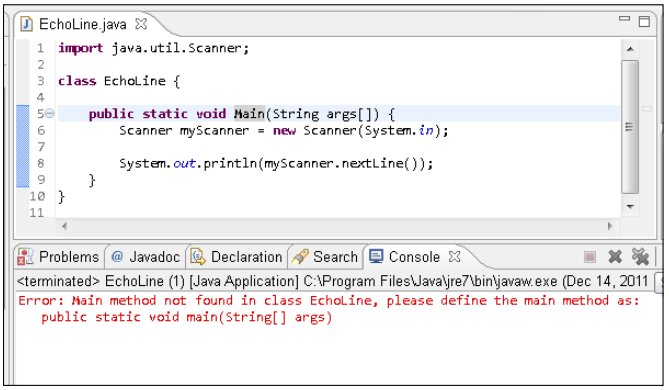


Figure 5-13: Whadaya mean "Main method not found in class EchoLine?"

Sure, your program has something named `Main`, but does it have anything named `main`? (Yes, I've heard of a famous poet named e. e. cummings, but who the heck is E. E. Cummings?) The computer doesn't presume that your word `Main` means the same thing as the expected word `main`. You need to change `Main` back to `main`. Then everything will be okay.

But in the meantime (or in the maintime), how does this improper capitalization make it past the compiler? Why don't you get any error messages when you compile the program? And if a capital `M` doesn't upset the compiler, why does this capital `M` mess everything up at runtime?

The answer goes back to the different kinds of words in the Java programming language. As it says in Chapter 4, Java has identifiers and keywords.

The keywords in Java are cast in stone. If you change `class` to `Class`, or change `public` to `Public`, then you get something new — something that the computer probably can't understand. That's why the compiler chokes on improper keyword capitalizations. It's the compiler's job to make sure that all the keywords are used properly.

On the other hand, the identifiers can bounce all over the place. Sure, there's an identifier named `main`, but you can make up a new identifier named `Main`. (You shouldn't do it, though. It's too confusing to people who know Java's usual meaning for the word `main`.) When the compiler sees a mistyped line, like `public static void Main`, the compiler just assumes that you're making up a brand-new name. So the compiler lets the line pass. You get no complaints from your old friend, the compiler.

But then, when you try to run the code, the computer goes ballistic. The Java virtual machine (JVM) runs your code. (For details, see Chapter 1.) The JVM needs to find a place to start executing statements in your code, so the JVM looks for a starting point named `main`, with a small `m`. If the JVM doesn't see anything named `main`, then the JVM gets upset. "Main method not found in class EchoLine," says the JVM. So at runtime, the JVM, and not the compiler, gives you an error message.

A better error message would be ***main*** *method not found in class EchoLine*, with a lowercase letter *m* in *main*. Here and there, the people who create the error messages overlook a detail or two.

What problem? I don't see a problem

I end this chapter on an upbeat note by showing you some of the things you can change in Listing 5-1 without rocking the boat.

The identifiers that you create

If you create an identifier, then that name is up for grabs. For example, in Listing 5-1, you can change `EchoLine` to `RepeatAfterMe`.

```
class RepeatAfterMe {

   public static void main ... etc.
```

This presents no problem at all, as long as you're willing to be consistent. Just follow most of the steps in this chapter's earlier "Typing and running a program" section.

- In Step 6, instead of typing **EchoLine**, type **RepeatAfterMe** in the New Java Class dialog box's Name field.
- In Step 8, when you copy the code from Listing 5-1, don't type

```
class EchoLine {
```

near the top of the listing. Instead, type the words

```
class RepeatAfterMe {
```

Spaces and indentation

Java isn't fussy about the use of spaces and indentation. All you need to do is keep your program well-organized and readable. Here's an alternative to spacing and indentation of the code in Listing 5-1:

```
import java.util.Scanner;
class EchoLine
{
  public static void main( String args[] )
  {
    Scanner myScanner =
        new Scanner( System.in );
    System.out.println
        ( myScanner.nextLine() );
  }
}
```

How you choose to do things

A program is like a fingerprint. No two programs look very much alike. Say I discuss a programming problem with a colleague. Then we go our separate ways and write our own programs to solve the same problem. Sure, we're duplicating the effort. But will we create the exact same code? Absolutely not. Everyone has his or her own style, and everyone's style is unique.

I asked fellow Java programmer David Herst to write his own `EchoLine` program without showing him my code from Listing 5-1. Here's what he wrote:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class EchoLine {
   public static void main(String[] args)
         throws IOException {
      InputStreamReader isr =
         new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
```

```
        String input = br.readLine();
        System.out.println(input);
    }
}
```

Don't worry about `BufferedReader`, `InputStreamReader`, or things like that. Just notice that, like snowflakes, no two programs are written exactly alike, even if they accomplish the same task. That's nice. It means your code, however different, can be as good as the next person's. That's very encouraging.

Chapter 6

Using the Building Blocks: Variables, Values, and Types

In This Chapter

- Declaring variables
- Assigning values to variables
- Working with numbers
- Using Java types

Back in 1946, John von Neumann wrote a groundbreaking paper about the newly emerging technology of computers and computing. Among other things, he established one fundamental fact: For all their complexity, the main business of computers is to move data from one place to another. Take a number — the balance in a person's bank account. Move this number from the computer's memory to the computer's processing unit. Add a few dollars to the balance and then move it back to the computer's memory. The movement of data . . . that's all there is; there ain't no more.

Good enough! This chapter shows you how to move around your data.

Using Variables

Here's an excerpt from a software company's website:

> SnitSoft recognizes its obligation to the information technology community. For that reason, SnitSoft is making its most popular applications available for a nominal charge. For just $5.95 plus shipping and handling, you receive a CD-ROM containing SnitSoft's premier products.

Go ahead. Click the Order Now! link. Just see what happens. You get an order form with two items on it. One item is labeled `$5.95 (CD-ROM)`, and the other item reads `$25.00 (shipping and handling)`. What a rip-off! Thanks to SnitSoft's generosity, you can pay $30.95 for 10 ten cents' worth of software.

Behind the scenes of the SnitSoft web page, a computer program does some scoundrel's arithmetic. The program looks something like the code in Listing 6-1.

Listing 6-1: SnitSoft's Grand Scam

```
class SnitSoft {

    public static void main(String args[]) {
        double amount;

        amount = 5.95;
        amount = amount + 25.00;

        System.out.print("We will bill $");
        System.out.print(amount);
        System.out.println(" to your credit card.");
    }
}
```

When I run Listing 6-1 code on my own computer (not the SnitSoft computer), I get the output shown in Figure 6-1.

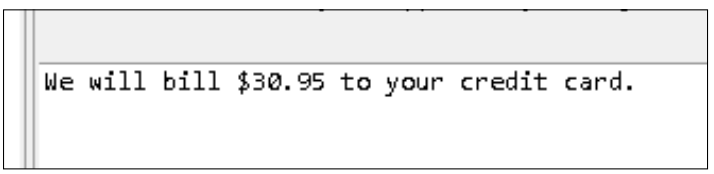


Figure 6-1: Running the code from Listing 6-1.

Using a variable

The code in Listing 6-1 makes use of a variable named `amount`. A *variable* is a placeholder. You can stick a number like 5.95 into a variable. After you've placed a number in the variable, you can change your mind and put a different number, like 30.95, into the variable. (That's what varies in a variable.) Of course, when you put a new number in a variable, the old number is no longer there. If you didn't save the old number somewhere else, the old number is gone.

Figure 6-2 gives a before-and-after picture of the code in Listing 6-1. When the computer executes `amount = 5.95`, the variable `amount` has the number 5.95 in it. Then, after the `amount = amount + 25.00` statement is executed, the variable `amount` suddenly has 30.95 in it. When you think about a variable, picture a place in the computer's memory where wires and transistors store 5.95, 30.95, or whatever. In Figure 6-2, imagine that each box is surrounded by millions of other such boxes.

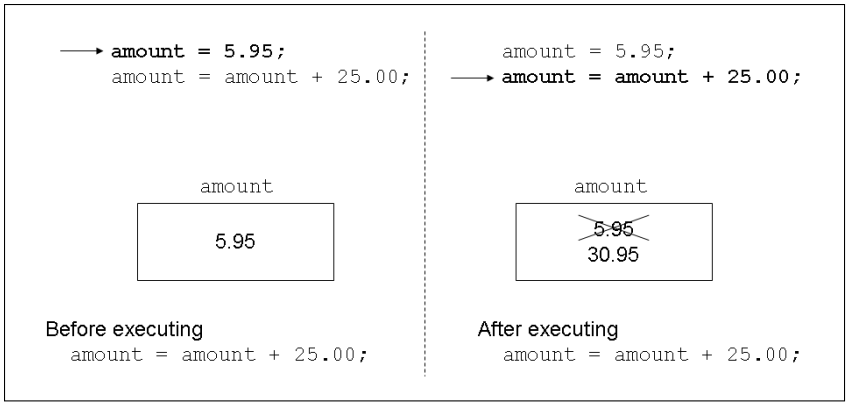


Figure 6-2: A variable (before and after).

Now you need some terminology. (You can follow along in Figure 6-3.) The thing stored in a variable is called a *value*. A variable's value can change during the run of a program (when SnitSoft adds the shipping and handling cost, for example). The value stored in a variable isn't necessarily a number. (You can, for example, create a variable that always stores a letter.) The kind of value stored in a variable is a variable's *type*. (You can read more about types in the rest of this chapter and in the next two chapters as well.)

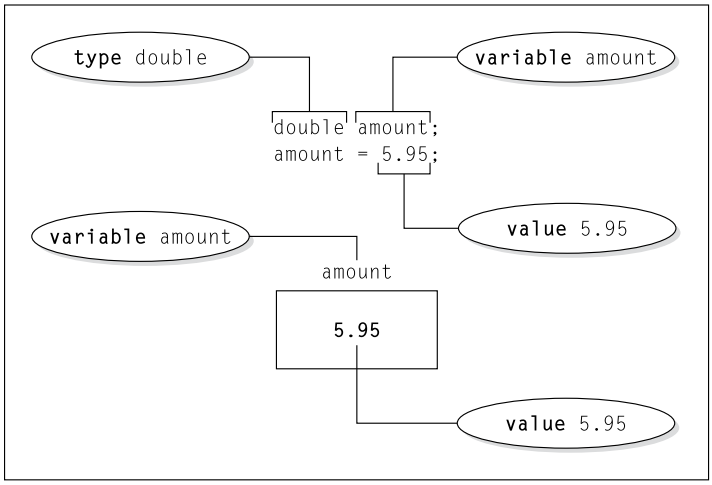


Figure 6-3: A variable, its value, and its type.

There's a subtle, almost unnoticeable difference between a variable and a variable's *name*. Even in formal writing, I often use the word *variable* when I mean *variable name*. Strictly speaking, `amount` is the variable name, and all the memory storage associated with `amount` (including the value and type of

`amount`) is the variable itself. If you think this distinction between *variable* and *variable name* is too subtle for you to worry about, join the club.

Every variable name is an identifier — a name that you can make up in your own code (for more about this, see Chapter 4). In preparing Listing 6-1, I made up the name `amount`.

Understanding assignment statements

The statements with equal signs in Listing 6-1 are called *assignment statements*. In an assignment statement, you assign a value to something. In many cases, this something is a variable.

You should get into the habit of reading assignment statements from right to left. For example, the first assignment statement in Listing 6-1 says, "Assign `5.95` to the `amount` variable." The second assignment statement is just a bit more complicated. Reading the second assignment statement from right to left, you get "Add `25.00` to the value that's already in the `amount` variable and make that number (`30.95`) be the new value of the `amount` variable." For a graphic, hit-you-over-the-head illustration of this, see Figure 6-4.

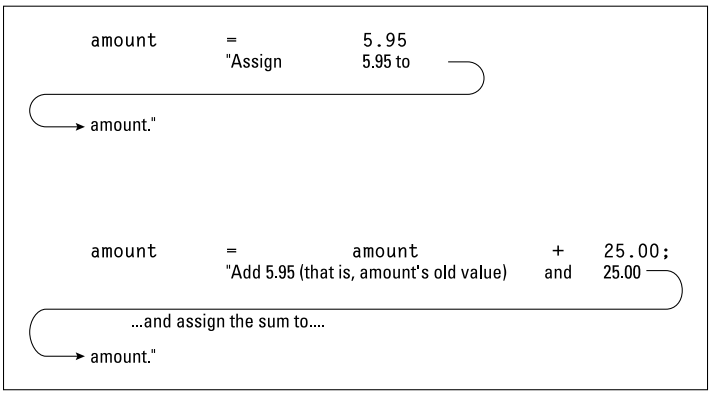


Figure 6-4: Reading an assignment statement from right to left.

In an assignment statement, the thing being assigned a value is always on the left side of the equal sign.

To wrap or not to wrap?

The last three statements in Listing 6-1 use a neat trick. You want the program to display just one line on the screen, but this line contains three different things:

- The line starts with `We will bill $`.
- The line continues with the `amount` variable's value.
- The line ends with `to your credit card`.

These are three separate things, so you put these things in three separate statements. The first two statements are calls to `System.out.print`. The last statement is a call to `System.out.println`.

Calls to `System.out.print` display text on part of a line and then leave the cursor at the end of the current line. After executing `System.out.print`, the cursor is still at the end of the same line, so the next `System.out.`*`whatever`* can continue printing on that same line. With several calls to `print` capped off by a single call to `println`, the result is just one nice-looking line of output, as Figure 6-5 illustrates.

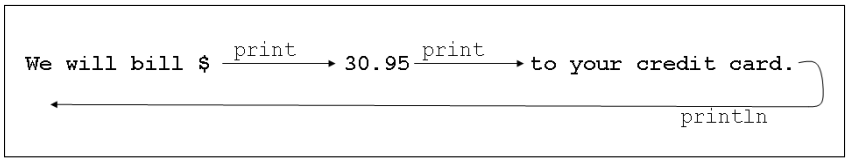


Figure 6-5: The roles played by `System.out.print` and `System.out.println`.

A call to `System.out.print` writes some things and leaves the cursor sitting at the end of the line of output. A call to `System.out.println` writes things and then finishes the job by moving the cursor to the start of a brand-new line of output.

What Do All Those Zeros and Ones Mean?

Here's a word:

gift

The question for discussion is, what does that word mean? Well, it depends on who looks at the word. For example, an English-speaking reader would say that "gift" stands for something one person bestows upon another in a box covered in bright paper and ribbons.

Look! I'm giving you a **gift**!

But in German, the word "gift" means "poison."

Let me give you some **gift**, my dear.

And in Swedish, "gift" can mean either "married" or "poison."

As soon as they got **gift**, she slipped a **gift** into his drink.

Then there's French. In France, there's a candy bar named "Gift."

He came for the holidays, and all he gave me was a bar of **Gift**.

So what do the letters g-i-f-t really mean? Well, they don't mean anything until you decide on a way to interpret them. The same is true of the zeros and ones inside a computer's circuitry.

Take, for example, the sequence 01001010. This sequence can stand for the letter *J*, but it can also stand for the number 74. That same sequence of zeros and ones can stand for $1.0369608636003646 \times 10^{-43}$. And when interpreted as screen pixels, the same sequence can represent the dots shown in Figure 6-6. The meaning of 01001010 depends entirely on the way the software interprets this sequence.

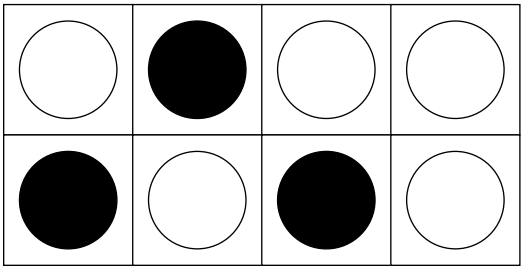


Figure 6-6: An extreme close-up of eight black-and-white screen pixels.

Types and declarations

How do you tell the computer what 01001010 stands for? The answer is in the concept called *type*. The type of a variable describes the kinds of values that the variable is permitted to store.

In Listing 6-1, look at the first line in the body of the `main` method:

```
double amount;
```

This line is called a *variable declaration*. Putting this line in your program is like saying, "I'm declaring my intention to have a variable named `amount` in my program." This line reserves the name `amount` for your use in the program.

In this variable declaration, the word *double* is a Java keyword. This word `double` tells the computer what kinds of values you intend to store in `amount`. In particular, the word `double` stands for numbers between -1.8×10^{308} and 1.8×10^{308}. That's an enormous range of numbers. Without the fancy $\times10$ notation, the second of these numbers is

```
18000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000.0
```

If the folks at SnitSoft ever charge that much for shipping and handling, they can represent the charge with a variable of type `double`.

What's the point?

More important than the humongous range of the `double` keyword's numbers is the fact that a `double` value can have digits to the right of the decimal point. After you declare `amount` to be of type `double`, you can store all sorts of numbers in `amount`. You can store 5.95, 0.02398479, or –3.0. In Listing 6-1, if I hadn't declared `amount` to be of type `double`, then I may not have been able to store 5.95. Instead, I would have had to store plain old 5 or dreary old 6, without any digits beyond the decimal point.

For more info on numbers without decimal points, see Chapter 7.

This paragraph deals with a really picky point, so skip it if you're not in the mood. People often use the phrase "decimal number" to describe a number with digits to the right of the decimal point. The problem is, the syllable "dec" stands for the number 10, so the word "decimal" implies a base-10 representation. Because computers store base-2 (not base-10) representations, the word "decimal" to describe such a number is a misnomer. But in this book, I just can't help myself. I'm calling them "decimal numbers" whether the techies like it or not.

Reading Decimal Numbers from the Keyboard

I don't believe it! SnitSoft is having a sale! For one week only, you can get the SnitSoft CD-ROM for the low price of just $5.75! Better hurry up and order one.

No, wait! Listing 6-1 has the price fixed at $5.95. I have to revise the program.

I know. I'll make the code more versatile. I'll input the amount from the keyboard. Listing 6-2 has the revised code, and Figure 6-7 shows a run of the new code.

Listing 6-2: Getting a Double Value from the Keyboard

```
import java.util.Scanner;

class VersatileSnitSoft {

   public static void main(String args[]) {
      Scanner myScanner = new Scanner(System.in);
      double amount;

      System.out.print("What's the price of a CD-ROM? ");
      amount = myScanner.nextDouble();
      amount = amount + 25.00;

      System.out.print("We will bill $");
      System.out.print(amount);
      System.out.println(" to your credit card.");
   }
}
```

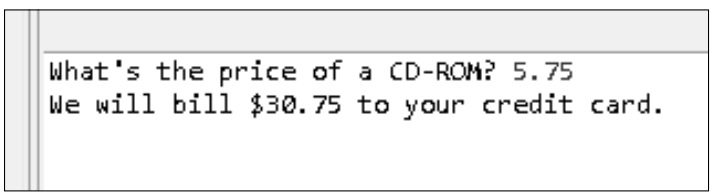


Figure 6-7: Getting the value of a double variable.

Though these be methods, yet there is madness in 't

Notice the call to the `nextDouble` method in Listing 6-2. Back in Listing 5-1, I use `nextLine`, but here in Listing 6-2, I use `nextDouble`.

Who does what, and how?

When you write a program, you're called a *programmer,* but when you run a program, you're called a *user*. So when you test your own code, you're being both the programmer and the user.

Suppose that your program contains a `myScanner.next`*`Something`*`()` call, like the calls in Listings 5-1 and 6-2. Then your program gets input from the user. But, when the program runs, how does the user know to type something on the keyboard? If the user and the programmer are the same person, and the program is fairly simple, then knowing what to type is no big deal. For example, when you start running the code in Listing 5-1, you have this book in front of you, and the book says "The computer is waiting for you to type something . . . You type one line of text . . ." So you type the text and press Enter. Everything is fine.

But very few programs come with their own books. In many instances, when a program starts running, the user has to stare at the screen to figure out what to do next. The code in Listing 6-2 works in this stare-at-the-screen scenario. In Listing 6-2, the first call to `print` puts an informative message (`What's the price of a CD-ROM?`) on the user's screen. A message of this kind is called a *prompt*.

When you start writing programs, you can easily confuse the roles of the prompt and the user's input. So remember, no preordained relationship exists between a prompt and the subsequent input. To create a prompt, you call `print` or `println`. Then, to read the user's input, you call `nextLine`, `nextDouble`, or one of the Scanner class's other `next`*`Something`* methods. These `print` and `next` calls belong in two separate statements. Java has no commonly used, single statement that does both the prompting and the "`next`-ing."

As the programmer, your job is to combine the prompting and the `next`-ing. You can combine prompting and `next`-ing in all kinds of ways. Some ways are helpful to the user, and some ways aren't.

- **If you don't have a call to `print` or `println`, then the user sees no prompt.** A blinking cursor sits quietly and waits for the user to type something. The user has to guess what kind of input to type. Occasionally that's okay, but usually it isn't.
- **If you call `print` or `println`, but you don't call a `myScanner.next`*`Something`* method, then the computer doesn't wait for the user to type anything.** The program races to execute whatever statement comes immediately after the `print` or `println`.
- **If your prompt displays a misleading message, then you mislead the user.** Java has no built-in feature that checks the appropriateness of a prompt. That's not surprising. Most computer languages have no prompt-checking feature.

So be careful with your prompts and gets. Be nice to your user. Remember, you were once a humble computer user, too.

In Java, each type of input requires its own special method. If you're getting a line of text, then `nextLine` works just fine. But if you're reading stuff from the keyboard, and you want that stuff to be interpreted as a number, you need a method like `nextDouble`.

To go from Listing 6-1 to Listing 6-2, I added an `import` declaration and some stuff about `new Scanner(System.in)`. You can find out more about these

things by reading the section on input and output in Chapter 5. (You can find out even more about input and output by visiting Chapter 13.) And more examples (more `myScanner.next`*`Something`* methods) are in Chapters 7 and 8.

Methods and assignments

Note how I use `myScanner.nextDouble` in Listing 6-2. The method `myScanner.nextDouble` is called as part of assignment statement. If you look in Chapter 5 at the section on how the EchoLine program works, you see that the computer can substitute something in place of a method call. The computer does this in Listing 6-2. When you type `5.75` on the keyboard, the computer turns

```
amount = myScanner.nextDouble();
```

into

```
amount = 5.75;
```

(The computer doesn't really rewrite the code in Listing 6-2. This `amount = 5.75` line just illustrates the effect of the computer's action.) In the second assignment statement in Listing 6-2, the computer adds 25.00 to the 5.75 that's stored in `amount`.

Some method calls have this substitution effect, and others (like `System.out.println`) don't. To find out more about this, see Chapter 19.

Variations on a Theme

In Listing 6-1, it takes two lines to give the `amount` variable its first value:

```
double amount;
amount = 5.95;
```

You can do the same thing with just one line:

```
double amount=5.95;
```

When you do this, you don't say that that you're "assigning" a value to the `amount` variable. The line `double amount=5.95` isn't called an "assignment statement." Instead, this line is called a declaration with an *initialization*.

You're *initializing* the `amount` variable. You can do all sorts of things with initializations, even arithmetic.

```
double gasBill  = 174.59;
double elecBill =  84.21;
double H2OBill  =  22.88;
double total    = gasBill + elecBill + H2OBill;
```

Moving variables from place to place

It helps to remember the difference between initializations and assignments. For one thing, you can drag a declaration with its initialization outside of a method.

```
//This is okay:
class SnitSoft {
    static double amount = 5.95;

    public static void main(String args[]) {
        amount = amount + 25.00;

        System.out.print("We will bill $");
        System.out.print(amount);
        System.out.println(" to your credit card.");
    }
}
```

You can't do the same thing with assignment statements. (See the following code and Figure 6-8.)

```
//This does not compile:
class BadSnitSoftCode {
    static double amount;

    amount = 5.95;          //Misplaced statement

    public static void main(String args[]) {
        amount = amount + 25.00;

        System.out.print("We will bill $");
        System.out.print(amount);
        System.out.println(" to your credit card.");
    }
}
```

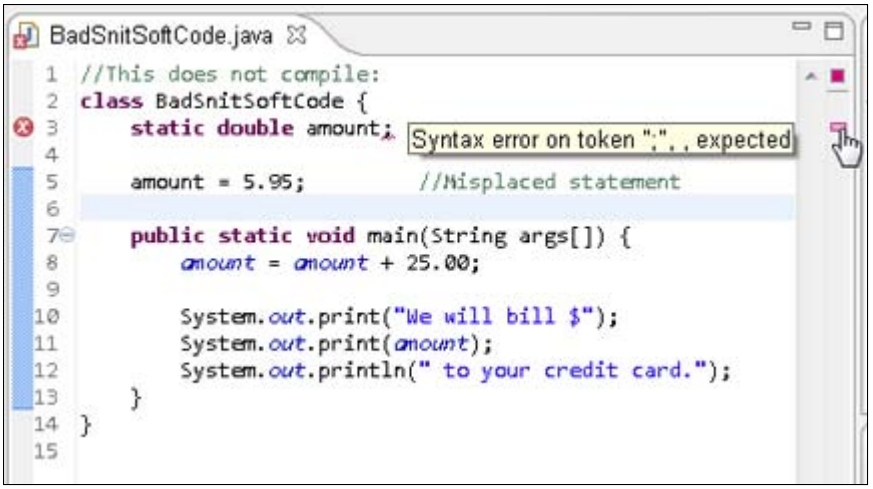


Figure 6-8: A failed attempt to compile BadSnit SoftCode.

You can't drag statements outside of methods. (Even though a variable declaration ends with a semicolon, a variable declaration isn't considered to be a statement. Go figure!)

The advantage of putting a declaration outside of a method is illustrated in Chapter 19. While you wait impatiently to reach that chapter, notice how I added the word `static` to each declaration that I pulled out of the `main` method. I had to do this because the `main` method's header has the word `static` in it. Not all methods are `static`. In fact, most methods aren't `static`. But whenever you pull a declaration out of a `static` method, you have to add the word `static` at the beginning of the declaration. All the mystery surrounding the word `static` is resolved in Chapter 18.

Combining variable declarations

The code in Listing 6-1 has only one variable (as if variables are in short supply). You can get the same effect with several variables.

```
class SnitSoftNew {

    public static void main(String args[]) {
        double cdPrice;
        double shippingAndHandling;
        double total;

        cdPrice = 5.95;
```

```
        shippingAndHandling = 25.00;
        total = cdPrice + shippingAndHandling;

        System.out.print("We will bill $");
        System.out.print(total);
        System.out.println(" to your credit card.");
    }
}
```

This new code gives you the same output as the code in Listing 6-1. (Refer to Figure 6-1.)

The new code has three declarations — one for each of the program's three variables. Because all three variables have the same type (the type `double`), I can modify the code and declare all three variables in one fell swoop:

```
double cdPrice, shippingAndHandling, total;
```

So which is better, one declaration or three declarations? Neither is better. It's a matter of personal style.

You can even add initializations to a combined declaration. When you do, each initialization applies to only one variable. For example, with the line

```
double cdPrice, shippingAndHandling = 25.00, total;
```

the value of `shippingAndHandling` becomes 25.00, but the variables `cdPrice` and `total` get no particular value.

Chapter 7

Numbers and Types

In This Chapter

- Processing whole numbers
- Making new values from old values
- Understanding Java's more exotic types

Not so long ago, people thought computers did nothing but big, number-crunching calculations. Computers solved arithmetic problems, and that was the end of the story.

In the 1980s, with the widespread use of word-processing programs, the myth of the big metal math brain went by the wayside. But even then, computers made great calculators. After all, computers are very fast and very accurate. Computers never need to count on their fingers. Best of all, computers don't feel burdened when they do arithmetic. I hate ending a meal in a good restaurant by worrying about the tax and tip, but computers don't mind that stuff at all. (Even so, computers seldom go out to eat.)

Using Whole Numbers

Let me tell you, it's no fun being an adult. Right now I have four little kids in my living room. They're all staring at me because I have a bag full of gumballs in my hand. With 30 gumballs in the bag, the kids are all thinking "Who's the best? Who gets more gumballs than the others? And who's going to be treated unfairly?" They insist on a complete, official gumball count, with each kid getting exactly the same number of tasty little treats. I must be careful. If I'm not, then I'll never hear the end of it.

With 30 gumballs and 4 kids, there's no way to divide the gumballs evenly. Of course, if I get rid of a kid, then I can give 10 gumballs to each kid. The trouble is, gumballs are disposable; kids are not. So my only alternative is to divvy up what gumballs I can and dispose of the rest. "Okay, think quickly," I say to myself. "With 30 gumballs and 4 kids, how many gumballs can I promise to each kid?"

I waste no time in programming my computer to figure out this problem for me. When I'm finished, I have the code in Listing 7-1.

Listing 7-1: How to Keep Four Kids from Throwing Tantrums

```
class KeepingKidsQuiet {

    public static void main(String args[]) {
        int gumballs;
        int kids;
        int gumballsPerKid;

        gumballs = 30;
        kids = 4;
        gumballsPerKid = gumballs / kids;

        System.out.print("Each kid gets ");
        System.out.print(gumballsPerKid);
        System.out.println(" gumballs.");
    }
}
```

A run of the `KeepingKidsQuiet` program is shown in Figure 7-1. If each kid gets seven gumballs, then the kids can't complain that I'm playing favorites. They'll have to find something else to squabble about.

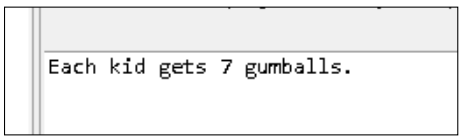


Figure 7-1: Fair and square.

At the core of the gumball problem, I've got whole numbers — numbers with no digits beyond the decimal point. When I divide 30 by 4, I get 7½, but I can't take the ½ seriously. No matter how hard I try, I can't divide a gumball in half, at least not without hearing "my half is bigger than his half." This fact is reflected nicely in Java. In Listing 7-1, all three variables (`gumballs`, `kids`, and `gumballsPerKid`) are of type `int`. An `int` value is a whole number. When you divide one `int` value by another (as you do with the slash in Listing 7-1), you get another `int`. When you divide 30 by 4, you get 7 — not 7½. You see this in Figure 7-1. Taken together, the statements

```
gumballsPerKid = gumballs/kids;

System.out.print(gumballsPerKid);
```

put the number `7` on the computer screen.

Reading whole numbers from the keyboard

What a life! Yesterday there were 4 kids in my living room, and I had 30 gumballs. Today there are 6 kids in my house, and I have 80 gumballs. How can I cope with all this change? I know! I'll write a program that reads the numbers of gumballs and kids from the keyboard. The program is in Listing 7-2, and a run of the program is shown in Figure 7-2.

Listing 7-2: A More Versatile Program for Kids and Gumballs

```
import java.util.Scanner;

class KeepingMoreKidsQuiet {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        int gumballs;
        int kids;
        int gumballsPerKid;

        System.out.print
            ("How many gumballs? How many kids? ");

        gumballs = myScanner.nextInt();
        kids = myScanner.nextInt();

        gumballsPerKid = gumballs / kids;

        System.out.print("Each kid gets ");
        System.out.print(gumballsPerKid);
        System.out.println(" gumballs.");
    }
}
```

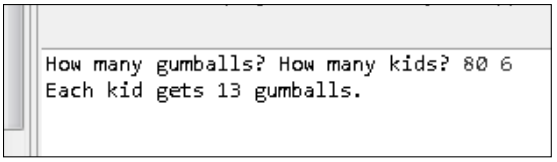


Figure 7-2: Next thing you know, I'll have 70 kids and 1,000 gumballs.

You should notice a couple of things about Listing 7-2. First, you can read an `int` value with the `nextInt` method. Second, you can issue successive calls to `Scanner` methods. In Listing 7-2, I call `nextInt` twice. All I have to do is

separate the numbers I type by blank spaces. In Figure 7-2, I put one blank space between my `80` and my `6`, but more blank spaces would work as well.

This blank space rule applies to many of the `Scanner` methods. For example, here's some code that reads three numeric values:

```
gumballs = myScanner.nextInt();
costOfGumballs = myScanner.nextDouble();
kids = myScanner.nextInt();
```

Figure 7-3 shows valid input for these three method calls.

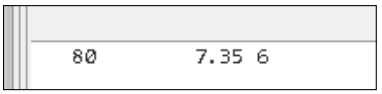


Figure 7-3: Three numbers for three `Scanner` method calls.

What you read is what you get

When you're writing your own code, you should never take anything for granted. Suppose that you accidentally reverse the order of the `gumballs` and `kids` assignment statements in Listing 7-2:

```
//This code is misleading:
System.out.print("How many gumballs? How many kids? ");

kids = myScanner.nextInt();
gumballs = myScanner.nextInt();
```

Then, the line `How many gumballs? How many kids?` is very misleading. Because the `kids` assignment statement comes before the `gumballs` assignment statement, the first number you type becomes the value of `kids`, and the second number you type becomes the value of `gumballs`. It doesn't matter that your program displays the message `How many gumballs? How many kids?`. What matters is the order of the assignment statements in the program.

If the `kids` assignment statement accidentally comes first, you can get a strange answer, like the zero answer in Figure 7-4. That's how `int` division works. It just cuts off any remainder. Divide a small number (like 6) by a big number (like 80), and you get 0.

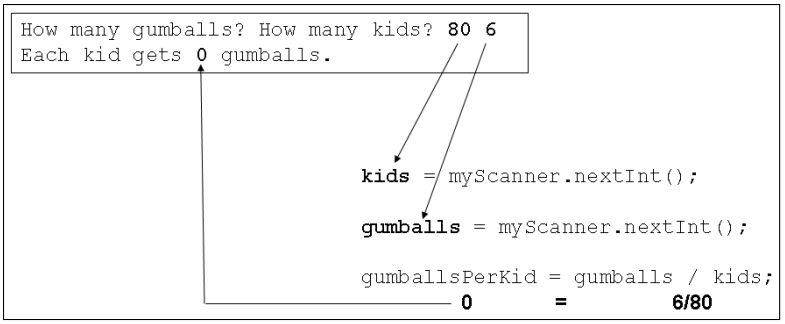

How many gumballs? How many kids? 80 6
Each kid gets 0 gumballs.
kids = myScanner.nextInt();
gumballs = myScanner.nextInt();
gumballsPerKid = gumballs / kids;
0 = 6/80

Figure 7-4: How to make six kids very unhappy.

Creating New Values by Applying Operators

What could be more comforting than your old friend, the plus sign? It was the first thing you learned about in elementary school math. Almost everybody knows how to add two and two. In fact, in English usage, adding two and two is a metaphor for something that's easy to do. Whenever you see a plus sign, one of your brain cells says, "Thank goodness, it could be something much more complicated."

So Java has a plus sign. You can use the plus sign to add two numbers:

```
int apples, oranges, fruit;
apples = 5;
oranges = 16;
fruit = apples + oranges;
```

Of course, the old minus sign is available, too:

```
apples = fruit - oranges;
```

Use an asterisk for multiplication and a forward slash for division:

```
double rate, pay, withholding;
int hours;

rate = 6.25;
hours = 35;
pay = rate * hours;
withholding = pay / 3.0;
```

When you divide an `int` value by another `int` value, you get an `int` value. The computer doesn't round. Instead, the computer chops off any remainder. If you put `System.out.println(11 / 4)` in your program, the computer prints `2`, not `2.75`. If you need a decimal answer, make either (or both) of the

numbers you're dividing `double` values. For example, if you put `System.out.println(11.0 / 4)` in your program, the computer divides a `double` value, `11.0`, by an `int` value, `4`. Because at least one of the two values is `double`, the computer prints `2.75`.

Finding a remainder

There's a useful arithmetic operator called the *remainder* operator. The symbol for the remainder operator is the percent sign (`%`). When you put `System.out.println(11 % 4)` in your program, the computer prints `3`. It does this because 4 goes into 11 who-cares-how-many times, with a remainder of 3.

Another name for the remainder operator is the *modulus* operator.

The remainder operator turns out to be fairly useful. After all, a remainder is the amount you have left over after you divide two numbers. What if you're making change for $1.38? After dividing 138 by 25, you have 13 cents left over, as shown in Figure 7-5.

The code in Listing 7-3 makes use of this remainder idea.

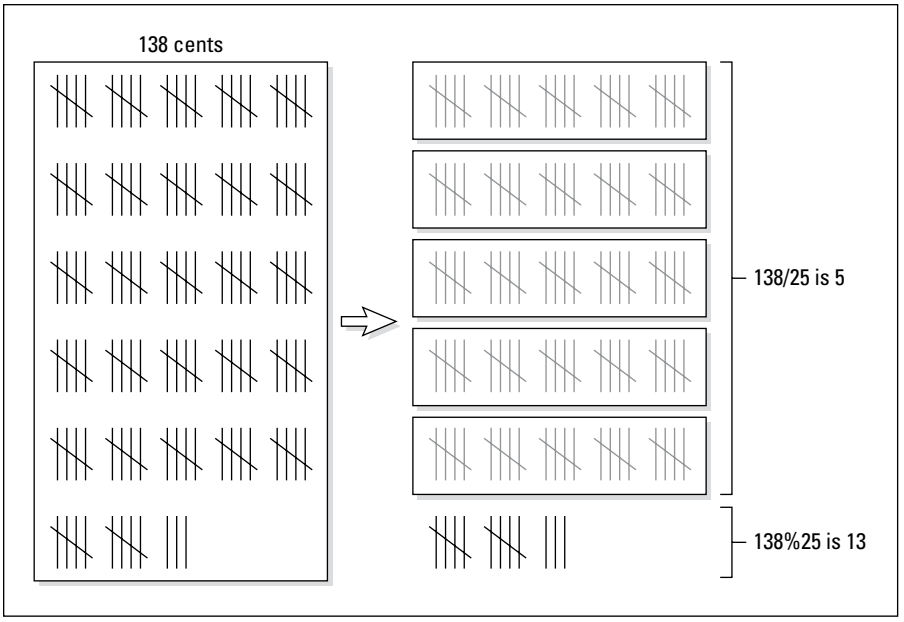


Figure 7-5: Hey, bud! Got change for 138 sticks?

Listing 7-3: Making Change

```
import java.util.Scanner;

class MakeChange {

   public static void main(String args[]) {
      Scanner myScanner = new Scanner(System.in);
      int quarters, dimes, nickels, cents;
      int whatsLeft, total;

      System.out.print("How many cents do you have? ");
      total = myScanner.nextInt();

      quarters = total / 25;
      whatsLeft = total % 25;

      dimes = whatsLeft / 10;
      whatsLeft = whatsLeft % 10;

      nickels = whatsLeft / 5;
      whatsLeft = whatsLeft % 5;

      cents = whatsLeft;

      System.out.println();
      System.out.println
            ("From " + total + " cents you get");
      System.out.println(quarters + " quarters");
      System.out.println(dimes + " dimes");
      System.out.println(nickels + " nickels");
      System.out.println(cents + " cents");
   }
}
```

A run of the code in Listing 7-3 is shown in Figure 7-6. You start with a `total` of 138 cents. The statement

```
quarters = total / 25;
```

divides 138 by 25, giving 5. That means you can make 5 quarters from 138 cents. Next, the statement

```
whatsLeft = total % 25;
```

divides 138 by 25 again, and puts only the remainder, 13, into `whatsLeft`. Now you're ready for the next step, which is to take as many dimes as you can out of 13 cents.

You keep going like this until you've divided away all the nickels. At that point, the value of `whatsLeft` is just 3 (meaning 3 cents).

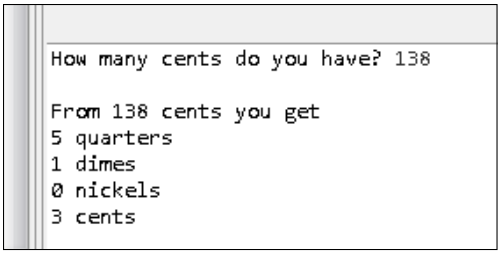


Figure 7-6: Change for $1.38.

If thine int offends thee, cast it out

The run in Figure 7-6 seems artificial. Why would you start with 138 cents? Why not use the more familiar $1.38? The reason is that the number 1.38 isn't a whole number, and whole numbers are more accurate than other kinds of numbers. In fact, without whole numbers, the remainder operator isn't very useful. For example, the value of `1.38 % 0.25` is `0.1299999999999999`. All those nines are tough to work with. Imagine reading your credit card statement and seeing that you owe $0.1299999999999999. You'd probably pay $0.13 and let the credit card company keep the change. But after years of rounding numbers, the credit card company would make a fortune! Chapter 8 describes inaccuracies caused by using `double` values in a bit more detail.

Throughout this book, I illustrate Java's `double` type with programs about money. Many authors do the same thing. But for greater accuracy, you should avoid using `double` values for money. Instead, you should use `int` values or use the `long` values that I describe in the last section of this chapter. Even better, look up `BigInteger` and `BigDecimal` in Java's API documentation. These `Big`*`SomethingOrOther`* types are cumbersome to use, but they provide industrial-strength numeric range and accuracy.

Now what if you want to input `1.38`, then the program should take your 1.38 and turn it into 138 cents? How can you get your program do this?

My first idea is to multiply 1.38 by 100:

```
//This doesn't quite work.
double amount;
int total;
amount=myScanner.nextDouble();
total=amount*100;
```

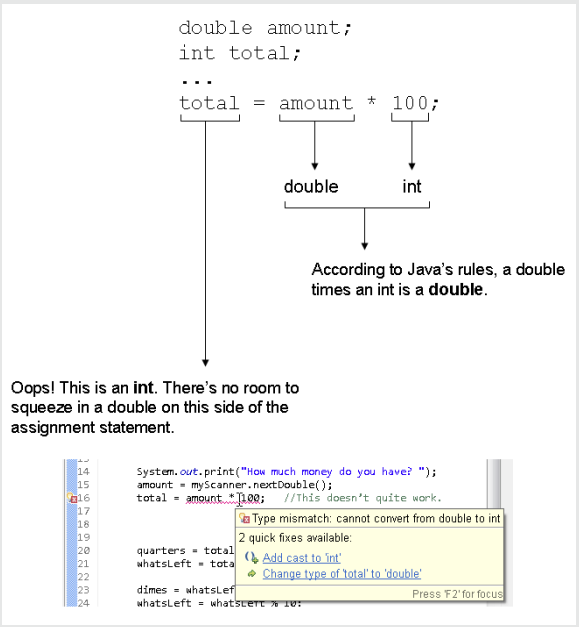
In everyday arithmetic, multiplying by 100 does the trick. But computers are fussy. With a computer, you have to be very careful when you mix `int` values and `double` values. (See the first figure in this sidebar.)

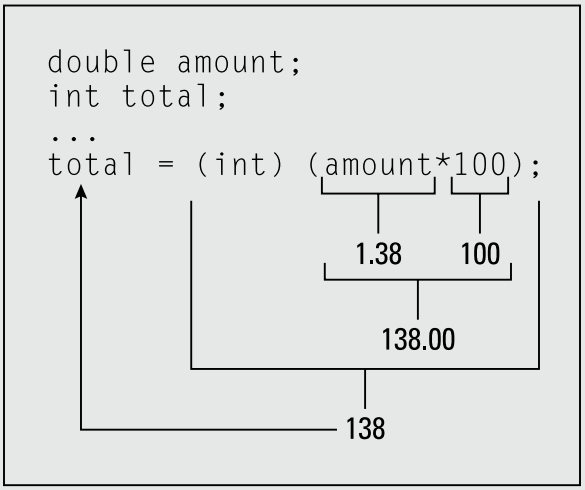
To cram a `double` value into an `int` variable, you need something called *casting*. When you cast a value, you essentially say. "I'm aware that I'm trying to squish a `double` value into an `int` variable. It's a tight fit, but I want to do it anyway."

To do casting, you put the name of a type in parentheses, as follows:

```
//This works!
total = (int) (amount * 100);
```

This casting notation turns the `double` value 138.00 into the `int` value 138, and everybody's happy. (See the second figure in this sidebar.)


double amount;
int total;
...
total = amount * 100;
double
int
According to Java's rules, a double times an int is a double.
Oops! This is an int. There's no room to squeeze in a double on this side of the assignment statement.
System.out.print("How much money do you have? ");
amount = myScanner.nextDouble();
total = amount * 100; //This doesn't quite work.
Type mismatch: cannot convert from double to int
2 quick fixes available:
Add cast to 'int'
Change type of 'total' to 'double'
Press 'F2' for focus



When two or more variables have similar types, you can create the variables with combined declarations. For example, Listing 7-3 has two combined declarations — one for the variables `quarters`, `dimes`, `nickels`, and `cents` (all of type `int`); another for the variables `whatsLeft` and `total` (both of type `int`). But to create variables of different types, you need separate declarations. For example, to create an `int` variable named `total` and a `double` variable named `amount`, you need one declaration `int total;` and another declaration `double amount;`.

Listing 7-3 has a call to `System.out.println()` with nothing in the parentheses. When the computer executes this statement, the cursor jumps to a new line on the screen. (I often use this statement to put a blank line in a program's output.)

The increment and decrement operators

Java has some neat little operators that make life easier (for the computer's processor, for your brain, and for your fingers). Altogether, there are four such operators — two increment operators and two decrement operators. The increment operators add one, and the decrement operators subtract one. To see how they work, you need some examples.

Using preincrement

The first example is in Figure 7-7.

A run of the program in Figure 7-7 is shown in Figure 7-8. In this horribly uneventful run, the count of gumballs gets displayed three times.

The double plus sign goes under two different names, depending on where you put it. When you put the `++` before a variable, the `++` is called the *preincrement* operator. In the word preincrement, the *pre* stands for *before.* In this setting, the word *before* has two different meanings:

- You're putting `++` before the variable.
- The computer adds 1 to the variable's value before the variable gets used in any other part of the statement.

Figure 7-9 has a slow-motion, instant replay of the preincrement operator's action. In Figure 7-9, the computer encounters the `System.out.println(++gumballs)` statement. First, the computer adds 1 to `gumballs` (raising the value of `gumballs` to 29). Then the computer executes `System.out.println`, using the new value of `gumballs` (29).

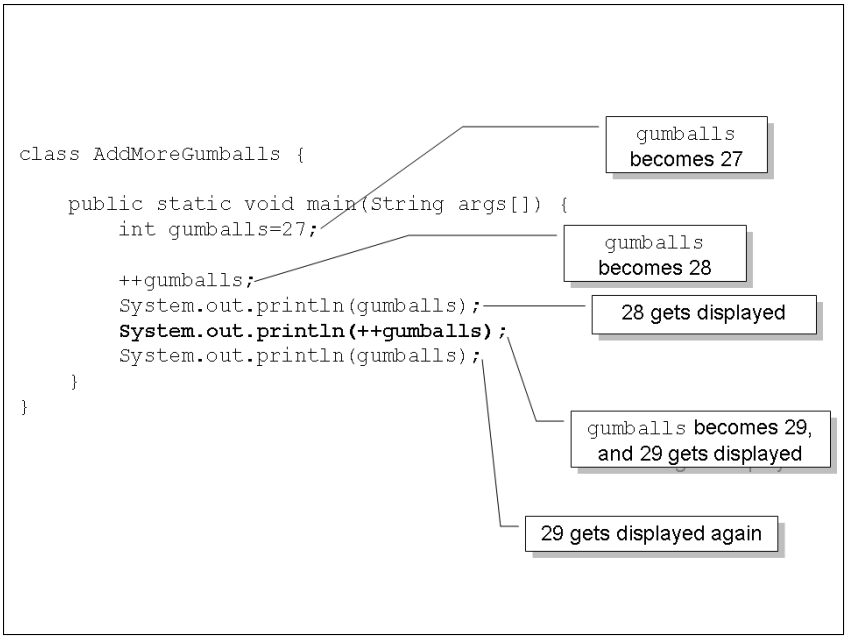


Figure 7-7: Using preincrement.

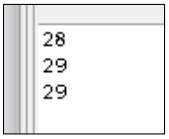


Figure 7-8: A run of the preincrement code (the code in Figure 7-7).

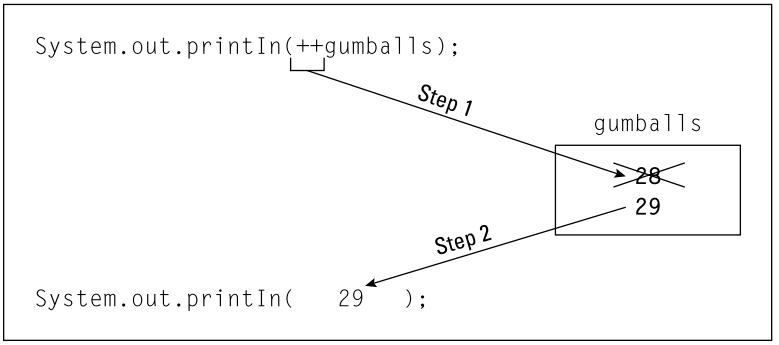


Figure 7-9: The preincrement operator in action.

With `System.out.println(++gumballs)`, the computer adds 1 to `gumballs` *before* printing the new value of `gumballs` on the screen.

Using postincrement

An alternative to preincrement is *postincrement*. With postincrement, the *post* stands for *after*. The word *after* has two different meanings:

- You put `++` after the variable.
- The computer adds 1 to the variable's value after the variable gets used in any other part of the statement.

Figure 7-10 has a close-up view of the postincrement operator's action. In Figure 7-10, the computer encounters the `System.out.println(gumballs++)` statement. First, the computer executes `System.out.println`, using the *old* value of `gumballs` (28). Then the computer adds 1 to `gumballs` (raising the value of `gumballs` to 29).

Look at the bold line of code in Figure 7-11. The computer prints the old value of `gumballs` (28) on the screen. Only after printing this old value does the computer add 1 to `gumballs` (raising the `gumballs` value from 28 to 29).

With `System.out.println(gumballs++)`, the computer adds 1 to `gumballs` *after* printing the old value that `gumballs` already had.

A run of the code in Figure 7-11 is shown in Figure 7-12. Compare Figure 7-12 with the run in Figure 7-8.

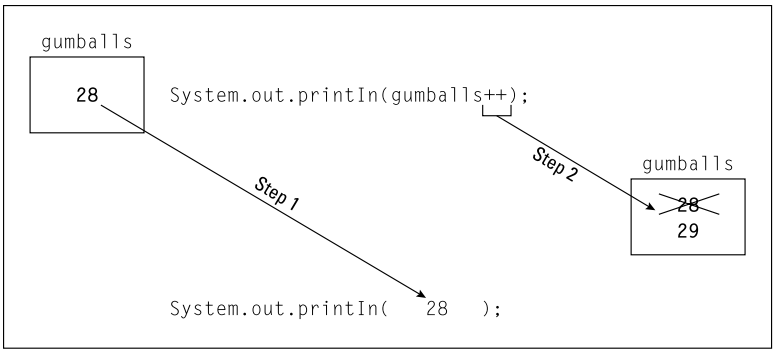


Figure 7-10: The post-increment operator in action.

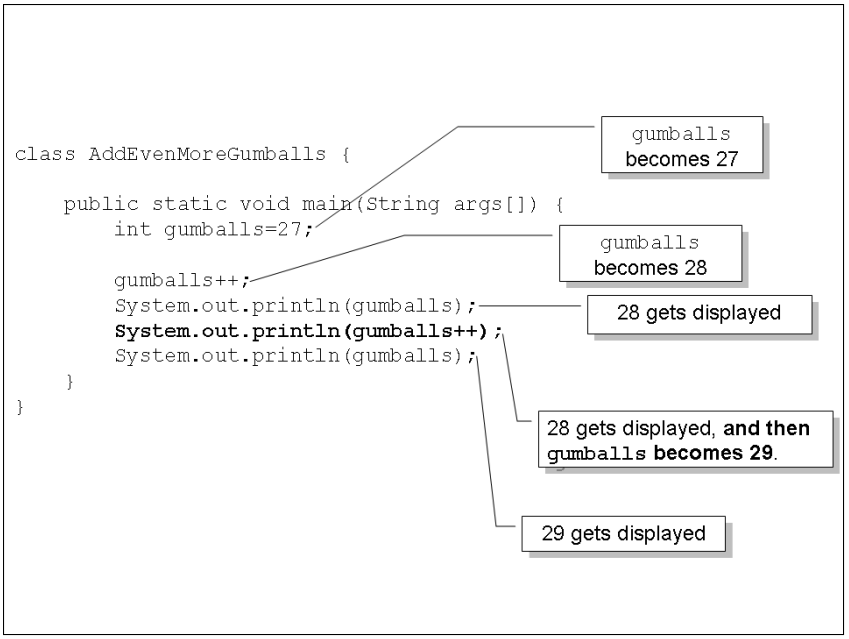


Figure 7-11: Using post-increment.

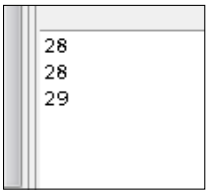


Figure 7-12: A run of the postincrement code (the code in Figure 7-11).

- With preincrement in Figure 7-8, the second number that gets displayed is `29`.
- With postincrement in Figure 7-12, the second number that gets displayed is `28`.

  In Figure 7-12, the number `29` doesn't show up on the screen until the end of the run, when the computer executes one last `System.out.println(gumballs)`.

Are you trying to decide between using preincrement or postincrement? Ponder no longer. Most programmers use postincrement. In a typical Java program, you often see things like `gumballs++`. You seldom see things like `++gumballs`.

Statements and expressions

Any part of a computer program that has a value is called an *expression*. If you write

```
gumballs = 30;
```

then `30` is an expression (an expression whose value is the quantity 30). If you write

```
amount = 5.95 + 25.00;
```

then `5.95 + 25.00` is an expression (because 5.95 + 25.00 has the value 30.95). If you write

```
gumballsPerKid =
      gumballs / kids;
```

then `gumballs / kids` is an expression. (The value of the expression `gumballs / kids` depends on whatever values the variables `gumballs` and `kids` have when the statement with the expression in it is executed.)

This brings us to the subject of the pre- and postincrement and decrement operators. There are two ways to think about these operators: the way everyone understands it and the right way. The way I explain it in most of this section (in terms of time, with *before* and *after*) is the way everyone understands the concept. Unfortunately, the way everyone understands the concept isn't really the right way. When you see `++` or `--`, you can think in terms of time sequence. But occasionally some programmer uses `++` or `--` in a convoluted way, and the notions of before and after break down. So if you're ever in a tight spot, you should think about these operators in terms of statements and expressions.

First, remember that a statement tells the computer to do something, and an expression has a value. (Statements are described in Chapter 4, and expressions are described earlier in this sidebar.) Which category does `gumballs++` belong to? The surprising answer is both. The Java code `gumballs++` is both a statement and an expression.

Suppose that, before executing the code `System.out.println(gumballs++)`, the value of `gumballs` is 28:

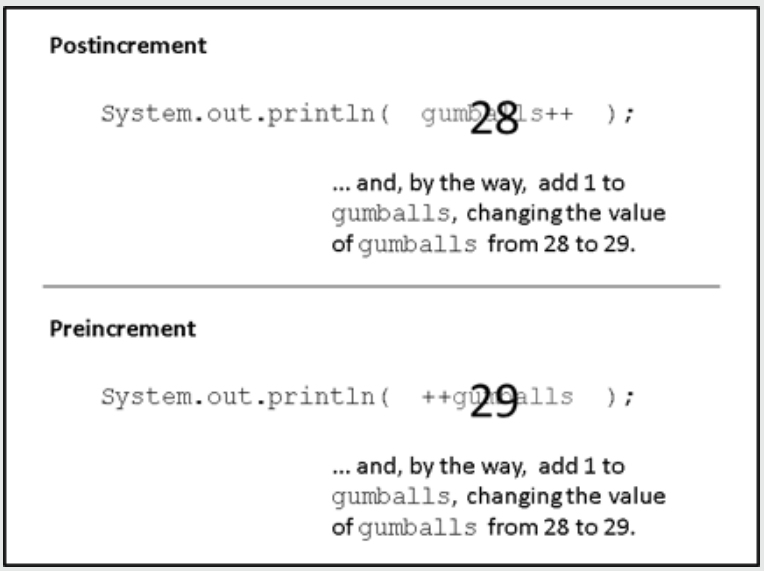
- As a statement, `gumballs++` tells the computer to add 1 to `gumballs`.
- As an expression, the value of `gumballs++` is 28, not 29.

So even though `gumballs` gets 1 added to it, the code `System.out.println(gumballs++)` really means `System.out.println(28)`. (See the figure in this sidebar.)

Now, almost everything you just read about `gumballs++` is true about `++gumballs`. The only difference is, as an expression, `++gumballs` behaves in a more intuitive way. Suppose that before executing the code `System.out.println(++gumballs)`, the value of `gumballs` is 28:

- As a statement, `++gumballs` tells the computer to add 1 to `gumballs`.
- As an expression, the value of `++gumballs` is 29.

So with `System.out.println(++gumballs)`, the variable `gumballs` gets 1 added to it, and the code `System.out.println(++gumballs)` really means `System.out.println(29)`.


Postincrement
System.out.println(gumballs++);
28
... and, by the way, add 1 to gumballs, changing the value of gumballs from 28 to 29.
Preincrement
System.out.println(++gumballs);
29
... and, by the way, add 1 to gumballs, changing the value of gumballs from 28 to 29.

In addition to preincrement and postincrement, Java has two operators that use `--`. These operators are called *predecrement* and *postdecrement:*

- With predecrement (`--gumballs`), the computer subtracts 1 from the variable's value before the variable gets used in the rest of the statement.
- With postdecrement (`gumballs--`), the computer subtracts 1 from the variable's value after the variable gets used in the rest of the statement.

Assignment operators

If you read the previous section — the section about operators that add 1 — you may be wondering if you can manipulate these operators to add 2, or add 5, or add 1000000. Can you write `gumballs++++` and still call yourself a Java programmer? Well, you can't. If you try it, then Eclipse will give you an error message:

```
Invalid argument to operation ++/--
```

If you don't use Eclipse, you may see a different error message:

```
unexpected type
required: variable
found   : value
      gumballs++++;
               ^
```

Eclipse or no Eclipse, the bottom line is the same: Namely, your code contains an error, and you have to fix it.

So how can you add values other than 1? As luck would have it, Java has plenty of *assignment operators* you can use. With an assignment operator, you can add, subtract, multiply, or divide by anything you want. You can do other cool operations, too.

For example, you can add 1 to the kids variable by writing

```
kids += 1;
```

Is this better than `kids++` or `kids = kids + 1`? No, it's not better. It's just an alternative. But you can add 5 to the `kids` variable by writing

```
kids += 5;
```

You can't easily add 5 with pre- or postincrement. And what if the kids get stuck in an evil scientist's cloning machine? The statement

```
kids *= 2;
```

multiplies the number of `kids` by 2.

With the assignment operators, you can add, subtract, multiply, or divide a variable by any number. The number doesn't have to be a literal. You can use a number-valued expression on the right side of the equal sign:

```
double amount = 5.95;
double shippingAndHandling = 25.00, discount = 0.15;

amount += shippingAndHandling;
amount -= discount * 2;
```

The preceding code adds 25.00 (`shippingAndHandling`) to the value of `amount`. Then, the code subtracts 0.30 (`discount * 2`) from the value of `amount`. How generous!

Size Matters

Here are today's new vocabulary words:

> **foregift** (fore-gift) *n.* A premium that a lessee pays to the lessor upon the taking of a lease.
>
> **hereinbefore** (here-in-be-fore) *adv.* In a previous part of this document.

Now imagine yourself scanning some compressed text. In this text, all blanks have been removed to conserve storage space. You come upon the following sequence of letters:

hereinbeforegiftedit

The question is, what do these letters mean? If you knew each word's length, you could answer the question.

here in be foregift edit

hereinbefore gifted it

herein before gift Ed it

A computer faces the same kind of problem. When a computer stores several numbers in memory or on a disk, the computer doesn't put blank spaces between the numbers. So imagine that a small chunk of the computer's memory looks like the stuff in Figure 7-13. (The computer works exclusively with zeros and ones, but Figure 7-13 uses ordinary digits. With ordinary digits, it's easier to see what's going on.)

What number or numbers are stored in Figure 7-13? Is it two numbers, 42 and 21? Or is it one number, 4,221? And what about storing four numbers, 4, 2, 2, and 1? It all depends on the amount of space each number consumes.

Figure 7-13: Storing the digits 4221.

Imagine a variable that stores the number of paydays in a month. This number never gets bigger than 31. You can represent this small number with just eight zeros and ones. But what about a variable that counts stars in the universe? That number could easily be more than a trillion, and to represent one trillion accurately, you need 64 zeros and ones.

At this point, Java comes to the rescue. Java has four types of whole numbers. Just as in Listing 7-1, I declare

```
int gumballsPerKid;
```

I can also declare

```
byte paydaysInAMonth;
short sickDaysDuringYourEmployment;
long numberOfStars;
```

Each of these types (`byte`, `short`, `int`, and `long`) has its own range of possible values (see Table 7-1).

Java has two types of decimal numbers (numbers with digits to the right of the decimal point). Just as in Listing 6-1, I declare

```
double amount;
```

I can also declare

```
float monthlySalary;
```

Given the choice between `double` and `float`, I always choose `double`. A variable of type `double` has a greater possible range of values and much greater accuracy (see Table 7-1).

Table 7-1 Java's Primitive Numeric Types

Type Name	*Range of Values*
Whole Number Types	
`byte`	–128 to 127
`short`	–32768 to 32767
`int`	–2147483648 to 2147483647
`long`	–9223372036854775808 to 9223372036854775807
Decimal Number Types	
`float`	-3.4×10^{38} to 3.4×10^{38}
`double`	-1.8×10^{308} to 1.8×10^{308}

Table 7-1 lists six of Java's *primitive* types (also known as *simple* types). Java has only eight primitive types, so only two of Java's primitive types are missing from Table 7-1.

Chapter 8 describes the two remaining primitive types. Chapter 17 introduces types that aren't primitive.

As a beginning programmer, you don't have to choose among the types in Table 7-1. Just use `int` for whole numbers and `double` for decimal numbers. If, in your travels, you see something like `short` or `float` in someone else's program, just remember the following:

- The types `byte`, `short`, `int`, and `long` represent whole numbers.
- The types `float` and `double` represent decimal numbers.

Most of the time, that's all you need to know.

Chapter 8

Numbers? Who Needs Numbers?

In This Chapter

- Working with characters
- Dealing with "true" or "false" values
- Rounding out your knowledge of Java's primitive types

I don't particularly like fax machines. They're so inefficient. Send a short fax, and what do you have? You have two slices of tree — one at the sending end, and another at the receiving end. You also have millions of dots — dots that scan tiny little lines across the printed page. The dots distinguish patches of light from patches of darkness. What a waste!

Compare a fax with an e-mail message. Using e-mail, I can send a 25-word contest entry with just 2,500 zeros and ones, and I don't waste any paper. Best of all, an e-mail message doesn't describe light dots and dark dots. An e-mail message contains codes for each of the letters — a short sequence of zeros and ones for the letter A, a different sequence of zeros and ones for the letter B, and so on. What could be simpler?

Now imagine sending a one-word fax. The word is "true," which is understood to mean, "true, I accept your offer to write *Beginning Programming with Java For Dummies,* 3rd Edition." A fax with this message sends a picture of the four letters t-r-u-e, with fuzzy lines where dirt gets on the paper and little white dots where the cartridge runs short on toner.

But really, what's the essence of the "true" message? There are just two possibilities, aren't there? The message could be "true" or "false," and to represent those possibilities, I need very little fanfare. How about 0 for "false" and 1 for "true?"

> They ask, "Do you accept our offer to write *Beginning Programming with Java For Dummies,* 3rd Edition?"
>
> "1," I reply.

Too bad I didn't think of that a few months ago. Anyway, this chapter deals with letters, truth, falsehood, and other such things.

Characters

In Chapters 6 and 7, you store numbers in all your variables. That's fine, but there's more to life than numbers. For example, I wrote this book with a computer, and this book contains thousands and thousands of non-numeric things called *characters*.

The Java type that's used to store characters is *char*. Listing 8-1 has a simple program that uses the `char` type, and a run of the Listing 8-1 program is shown in Figure 8-1.

Listing 8-1: Using the char Type

```
class LowerToUpper {

    public static void main(String args[]) {
        char smallLetter, bigLetter;

        smallLetter = 'b';
        bigLetter = Character.toUpperCase(smallLetter);
        System.out.println(bigLetter);
    }
}
```

Figure 8-1: Exciting program output!

In Listing 8-1, the first assignment statement stores the letter `b` in the `smallLetter` variable. In that statement, notice how `b` is surrounded by single quote marks. In a Java program, every `char` literal starts and ends with a single quote mark.

When you surround a letter with quote marks, you tell the computer that the letter isn't a variable name. For example, in Listing 8-1, the incorrect statement `smallLetter = b` would tell the computer to look for a variable named `b`. Because there's no variable named `b`, you'd get a `b cannot be resolved to a variable` message.

In the second assignment statement of Listing 8-1, the program calls an API method whose name is `Character.toUpperCase`. The method `Character.toUpperCase` does what its name suggests — the method produces the uppercase equivalent of a lowercase letter. In Listing 8-1, this uppercase equivalent (the letter `B`) is assigned to the variable `bigLetter`, and the `B` that's in `bigLetter` is printed on the screen, as illustrated in Figure 8-2.

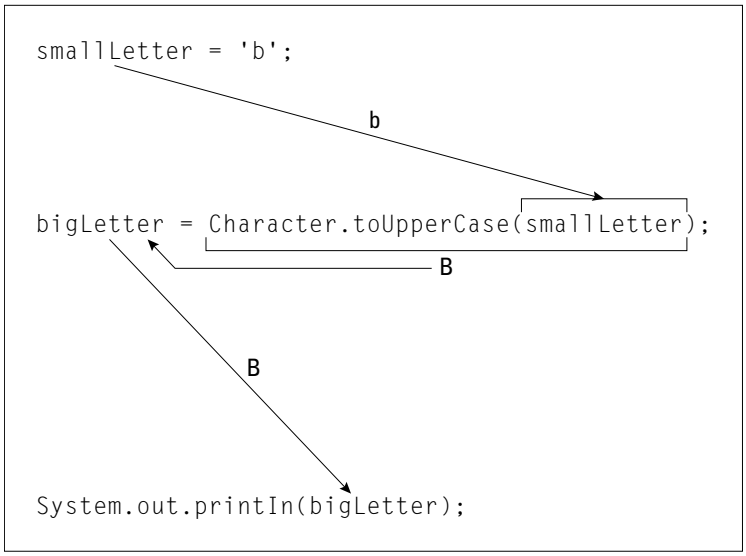


Figure 8-2: The action in Listing 8-1.

When the computer displays a `char` value on the screen, the computer does not surround the character with single quote marks.

I digress . . .

A while ago, I wondered what would happen if I called the `Character.toUpperCase` method and fed the method a character that isn't lowercase to begin with. I yanked out the Java API documentation, but I found no useful information. The documentation said that `toUpperCase` "converts the character argument to uppercase using case mapping information from the UnicodeData file." Thanks, but that's not useful to me.

Silly as it seems, I asked myself what I'd do if I were the `toUpperCase` method. What would I say if someone handed me a capital `R` and told me to capitalize that letter? I'd say, "Take back your stinking capital `R`." In the lingo of computing, I'd send that person an error message. So I wondered if I'd get an error message when I applied `Character.toUpperCase` to the letter `R`.

I tried it. I cooked up the experiment in Listing 8-2.

Listing 8-2: Investigating the Behavior of toUpperCase

```
class MyExperiment {

    public static void main(String args[]) {
        char smallLetter, bigLetter;

        smallLetter = 'R';
        bigLetter = Character.toUpperCase(smallLetter);
        System.out.println(bigLetter);

        smallLetter = '3';
        bigLetter = Character.toUpperCase(smallLetter);
        System.out.println(bigLetter);
    }
}
```

In my experiment, I didn't mix chemicals and blow things up. Here's what I did instead:

- **I assigned `'R'` to `smallLetter`.**

  The `toUpperCase` method took the uppercase `R` and gave me back another uppercase `R` (see Figure 8-3). I got no error message. This told me what the `toUpperCase` method does with a letter that's already uppercase. The method does nothing.

- **I assigned `'3'` to `smallLetter`.**

  The `toUpperCase` method took the digit `3` and gave me back the same digit `3` (see Figure 8-3). I got no error message. This told me what the `toUpperCase` method does with a character that's not a letter. It does nothing, zip, zilch, bupkis.

Figure 8-3: Running the code in Listing 8-2.

I write about this experiment to make an important point. When you don't understand something about computer programming, it often helps to write a test program. Make up an experiment and see how the computer responds.

I guessed that handing a capital `R` to the `toUpperCase` method would give me an error message, but I was wrong. See? The answers to questions aren't

handed down from heaven. The people who created the Java API made decisions. They made some obvious choices, and but they also made some unexpected choices. No one knows everything about Java's features, so don't expect to cram all the answers into your head.

The Java documentation is great, but for every question that the documentation answers, it ignores three other questions. So be bold. Don't be afraid to tinker. Write lots of short, experimental programs. You can't break the computer, so play tough with it. Your inquisitive spirit will always pay off.

Reading and understanding Java's API documentation is an art, not a science. For advice on making the most of these docs, take a look at the Appendix on this book's website — `http://allmycode.com/BeginProg3`.

One character only, please

A `char` variable stores only one character. So if you're tempted to write the following statements

```
char smallLetters;
smallLetters = 'barry';  //Don't do this
```

please resist the temptation. You can't store more than one letter at a time in a `char` variable, and you can't put more than one letter between a pair of single quotes. If you're trying to store words or sentences (not just single letters), then you need to use something called a *String*. For a look at Java's `String` type, see Chapter 18.

Variables and recycling

In Listing 8-2, I use `smallLetter` twice, and I use `bigLetter` twice. That's why they call these things *variables*. First, the value of `smallLetter` is `R`. Later, I vary the value of `smallLetter` so that the value of `smallLetter` becomes `3`.

When I assign a new value to `smallLetter`, the old value of `smallLetter` gets obliterated. For example, in Figure 8-4, the second `smallLetter` assignment puts `3` into `smallLetter`. When the computer executes this second assignment statement, the old value `R` is gone.

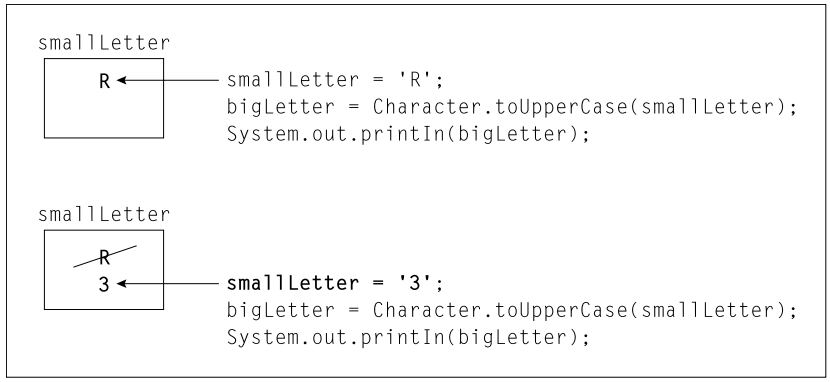


Figure 8-4: Varying the value of `smallLetter`.

Is that okay? Can you afford to forget the value that `smallLetter` once had? Yes, in Listing 8-2, it's okay. After you've assigned a value to `bigLetter` with the statement

```
bigLetter = Character.toUpperCase(smallLetter);
```

you can forget all about the existing `smallLetter` value. You don't need to do this:

```
// This code is cumbersome.
// The extra variables are unnecessary.
char smallLetter1, bigLetter1;
char smallLetter2, bigLetter2;

smallLetter1 = 'R';
bigLetter1 = Character.toUpperCase(smallLetter1);
System.out.println(bigLetter1);

smallLetter2 = '3';
bigLetter2 = Character.toUpperCase(smallLetter2);
System.out.println(bigLetter2);
```

You don't need to store the old and new values in separate variables. Instead, you can reuse the variables `smallLetter` and `bigLetter` as in Listing 8-2.

This reuse of variables doesn't save you from a lot of extra typing. It doesn't save much memory space, either. But reusing variables keeps the program uncluttered. When you look at Listing 8-2, you can see at a glance that the code has two parts, and you see that both parts do roughly the same thing.

The code in Listing 8-2 is simple and manageable. In such a small program, simplicity and manageability don't matter very much. But in a large program, it helps to think carefully about the use of each variable.

When not to reuse a variable

The previous section discusses the reuse of variables to make a program slick and easy to read. This section shows you the flip side. In this section, the problem at hand forces you to create new variables.

Suppose that you're writing code to reverse the letters in a four-letter word. You store each letter in its own separate variable. Listing 8-3 shows the code, and Figure 8-5 shows the code in action.

Listing 8-3: Making a Word Go Backward

```
import java.util.Scanner;

class ReverseWord {

  public static void main(String args[]) {
    Scanner myScanner = new Scanner(System.in);
    char c1, c2, c3, c4;

    c1 = myScanner.findWithinHorizon(".", 0).charAt(0);
    c2 = myScanner.findWithinHorizon(".", 0).charAt(0);
    c3 = myScanner.findWithinHorizon(".", 0).charAt(0);
    c4 = myScanner.findWithinHorizon(".", 0).charAt(0);

    System.out.print(c4);
    System.out.print(c3);
    System.out.print(c2);
    System.out.print(c1);
    System.out.println();
  }
}
```

Figure 8-5: Stop those pots!

The trick in Listing 8-3 is as follows:

- Assign values to variables `c1`, `c2`, `c3`, and `c4` in that order.
- Display these variables' values on the screen in reverse order: `c4`, `c3`, `c2`, and then `c1`, as illustrated in Figure 8-6.

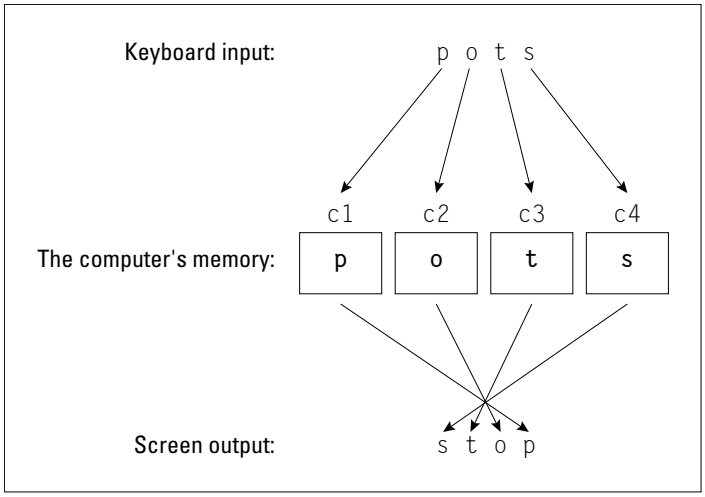


Figure 8-6: Using four variables.

If you don't use four separate variables, then you don't get the result that you want. For example, imagine that you store characters in only one variable. You run the program and type the word `pots`. When it's time to display the word in reverse, the computer remembers the final `s` in the word `pots`. But the computer doesn't remember the `p`, the `o`, or the `t`, as shown in Figure 8-7.

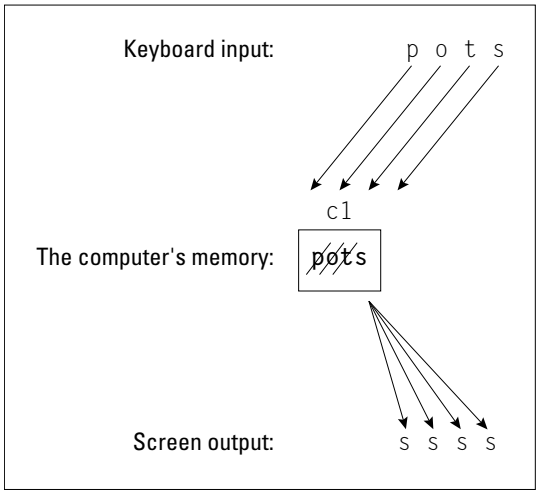


Figure 8-7: Getting things wrong because you used only one variable.

I wish I could give you 12 simple rules to help you decide when and when not to reuse variables. The problem is, I can't. It all depends on what you're trying to accomplish. So how do you figure out on your own when and when not to reuse variables? Like the guy says to the fellow who asks how to get to Carnegie Hall, "Practice, practice, practice."

What's behind all this findWithinHorizon nonsense?

Without wallowing in too much detail, here's how the `findWithinHorizon(".", 0).charAt(0)` technique works:

Java's `findWithinHorizon` method looks for things in the input. The things the method finds depend on the stuff you put in parentheses. For example, a call to `findWithinHorizon("\\d\\d\\d", 0)` looks for a group consisting of three digits. With the following line of code

```
System.out.println(myScanner.
  findWithinHorizon("\\d\\d\\d", 0));
```

I can type

```
Testing 123 Testing Testing
```

and the computer responds by displaying

```
123
```

In the call `findWithinHorizon("\\d\\d\\d", 0)`, each `\\d` stands for a single digit. This `\\d` business is one of many abbreviations in special code called *regular expressions*.

Now here's something strange. In the world of regular expressions, a dot stands for any character at all. (That is, a dot stands for "any character, not necessarily a dot.") So `findWithinHorizon(".", 0)` tells the computer to find the next character of any kind that the user types on the keyboard. When you're trying to input a single character, `findWithinHorizon(".", 0)` is mighty useful.

In the call `findWithinHorizon("\\d\\d\\d", 0)`, the `0` tells `findWithinHorizon` to keep searching until the end of the input. This value `0` is a special case because anything other than `0` limits the search to a certain number of characters. (That's why the method name contains the word *horizon*. The *horizon* is as far as the method sees.) Here are a few examples:

- With the same input `Testing 123 Testing Testing`, the call `findWithinHorizon("\\d\\d\\d", 9)` returns `null`. It returns `null` because the first nine characters of the input (the characters `Testing 1` — seven letters, a blank space, and a digit) don't contain three consecutive digits. These nine characters don't match the pattern `\\d\\d\\d`.
- With the same input, the call `findWithinHorizon("\\d\\d\\d", 10)` also returns `null`. It returns `null` because the first ten characters of the input (the characters `Testing 12`) don't contain three consecutive digits.
- With the same input, the call `findWithinHorizon("\\d\\d\\d", 11)` returns `123`. It returns `123` because the first 11 characters of the input (the characters `Testing 123`) contain these 3 consecutive digits.
- With the input `A57B442123 Testing`, the call `findWithinHorizon("\\d\\d\\d", 12)` returns `442`. It returns `442` because among the first 12 characters of the input (the characters `A57B442123 Test`), the first sequence consisting of 3 consecutive digits is the sequence `442`.

(continued)

(continued)

But wait! To grab a single character from the keyboard, I call `findWithinHorizon(".", 0).charAt(0)`. What's the role of `charAt(0)` in reading a single character? Unfortunately, any `findWithinHorizon` call behaves as if it's finding a bunch of characters, not just a single character. Even when you call `findWithinHorizon(".", 0)`, and the computer fetches just one letter from the keyboard, the Java program treats that letter as one of possibly many input characters.

The call to `charAt(0)` takes care of the multicharacter problem. This `charAt(0)` call tells Java to pick the initial character from any of the characters that `findWithinHorizon` fetches.

Yes, it's complicated. And yes, I don't like having to explain it. But no, you don't have to understand any of the details in this sidebar. Just read the details if you want to read them and skip the details if you don't care.

Reading characters

The people who created Java's `Scanner` class didn't create a `next` method for reading a single character. So to input a single character, I paste two Java API methods together. I use the `findWithinHorizon` and `charAt` methods.

Table 5-1 in Chapter 5 introduces this `findWithinHorizon(".", 0).charAt(0)` technique for reading a single input character, and Listing 8-3 uses the technique to read one character at a time. (In fact, Listing 8-3 uses the technique four times to read four individual characters.)

Notice the format for the input in Figure 8-5. To enter the characters in the word `pots`, I type four letters, one after another, with no blank spaces between the letters and no quote marks. The `findWithinHorizon(".", 0).charAt(0)` technique works that way, but don't blame me or my technique. Other developers' character-reading methods work the same way. No matter whose methods you use, reading a character differs from reading a number. Here's how:

- **With methods like `nextDouble` and `nextInt`, you type blank spaces between numbers.**

  If I type **80 6**, then two calls to `nextInt` read the number `80`, followed by the number `6`. If I type **806**, then a single call to `nextInt` reads the number `806` (eight hundred six), as illustrated in Figure 8-8.

- **With `findWithinHorizon(".", 0).charAt(0)`, you don't type blank spaces between characters.**

  If I type **po**, then two successive calls to `findWithinHorizon(".", 0).charAt(0)` read the letter `p`, followed by the letter `o`. If I type **p o**, then two calls to `findWithinHorizon(".", 0).charAt(0)` read the letter `p`, followed by a blank space character. (Yes, the blank space is a character!) Again, see Figure 8-8.

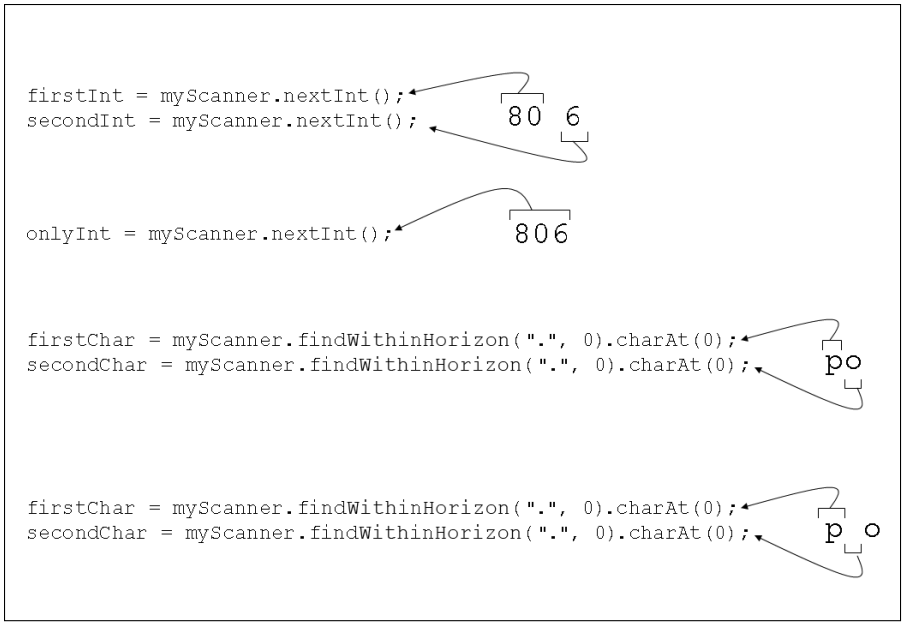


Figure 8-8: Reading numbers and characters.

To represent a lone character in the text of a computer program, you surround the character with single quote marks. But, when you type a character as part of a program's input, you don't surround the character with quote marks.

Suppose that your program calls `nextInt` and then `findWithinHorizon (".", 0).charAt(0)`. If you type **80x** on the keyboard, you get an error message. (The message says `InputMismatchException`. The `nextInt` method expects you to type a blank space after each `int` value.) Now what happens if, instead of typing **80x**, you type **80 x** on the keyboard? Then the program gets `80` for the `int` value, followed by a blank space for the character value. For the program to get the `x`, the program has to call `findWithin Horizon(".", 0).charAt(0)` one more time. It seems wasteful, but it makes sense in the long run.

The boolean Type

I'm in big trouble. I have 140 gumballs, and 15 kids are running around and screaming in my living room. They're screaming because each kid wants 10 gumballs, and they're running because that's what kids do in a crowded living room. I need a program that tells me if I can give 10 gumballs to each kid.

I need a variable of type *boolean*. A `boolean` variable stores one of 2 values — `true` or `false` (`true`, I can give 10 gumballs to each kid; or `false`, I can't give 10 gumballs to each kid). Anyway, the kids are going berserk, so I've written a short program and put it in Listing 8-4. The output of the program is shown in Figure 8-9.

Listing 8-4: Using the boolean Type

```
class CanIKeepKidsQuiet {

    public static void main(String args[]) {
        int gumballs;
        int kids;
        int gumballsPerKid;
        boolean eachKidGetsTen;

        gumballs = 140;
        kids = 15;
        gumballsPerKid = gumballs / kids;

        System.out.print("True or false? ");
        System.out.println("Each kid gets 10 gumballs.");
        eachKidGetsTen = gumballsPerKid >= 10;
        System.out.println(eachKidGetsTen);
    }
}
```

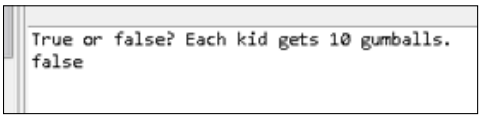


Figure 8-9: Oh, no!

In Listing 8-4, the variable `eachKidGetsTen` is of type `boolean`. So the value stored in the `eachKidGetsTen` variable can be either `true` or `false`. (I can't store a number or a character in the `eachKidGetsTen` variable.)

To find a value for the variable `eachKidGetsTen`, the program checks to see whether `gumballsPerKid` is greater than or equal to ten. (The symbols `>=` stand for "greater than or equal to." What a pity! There's no ≥ key on the standard computer keyboard.) Because `gumballsPerKid` is only nine, `gumballsPerKid >= 10` is false. So `eachKidGetsTen` becomes `false`. Yikes! The kids will tear the house apart! (Before they do, take a look at Figure 8-10.)

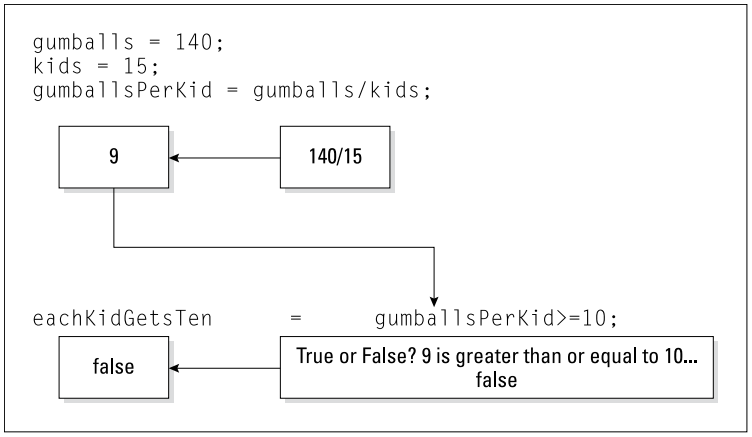


Figure 8-10: Assigning a value to the `eachKidGetsTen` variable.

Expressions and conditions

In Listing 8-4, the code `gumballsPerKid >= 10` is an expression. The expression's value depends on the value stored in the variable `gumballsPerKid`. On a bad day, the value of `gumballsPerKid >= 10` is `false`. So the variable `eachKidGetsTen` is assigned the value `false`.

An expression like `gumballsPerKid >= 10`, whose value is either `true` or `false`, is sometimes called a *condition*.

Values like `true` and `false` may look as if they contain characters, but they really don't. Internally, the Java Virtual Machine doesn't store `boolean` values with the letters t-r-u-e or f-a-l-s-e. Instead, the JVM stores codes, like 0 for false and 1 for true. When the computer displays a `boolean` value (as in `System.out.println(eachKidGetsTen)`), the Java virtual machine converts a code like 0 into the five-letter word `false`.

Comparing numbers; comparing characters

In Listing 8-4, I compare a variable's value with the number 10. I use the `>=` operator in the expression

```
gumballsPerKid >= 10
```

Of course, the greater-than-or-equal-to comparison gets you only so far. Table 8-1 shows you the operators you can use to compare things with one another.

Table 8-1 Comparison Operators

Operator Symbol	*Meaning*	*Example*
`==`	is equal to	`yourGuess == winningNumber`
`!=`	is not equal to	`5 != numberOfCows`
`<`	is less than	`strikes < 3`
`>`	is greater than	`numberOfBoxtops > 1000`
`<=`	is less than or equal to	`numberOfCows + numberOfBulls <= 5`
`>=`	is greater than or equal to	`gumballsPerKid >= 10`

With the operators in Table 8-1, you can compare both numbers and characters.

Notice the double equal sign in the first row of Table 8-1. Don't try to use a single equal sign to compare two values. The expression `yourGuess = winningNumber` (with a single equal sign) doesn't compare `yourGuess` with `winningNumber`. Instead, `yourGuess = winningNumber` changes the value of `yourGuess`. (It assigns the value of `winningNumber` to the variable `yourGuess`.)

You can compare other things (besides numbers and characters) with the `==` and `!=` operators. But when you do, you have to be careful. For more information, see Chapter 18.

Comparing numbers

Nothing is more humdrum than comparing numbers. "True or false? Five is greater than or equal to ten." False. Five is neither greater than nor equal to ten. See what I mean? Bo-ring.

Comparing whole numbers is an open-and-shut case. But unfortunately, when you compare decimal numbers, there's a wrinkle. Take a program for converting from Celsius to Fahrenheit. Wait! Don't take just any such program; take the program in Listing 8-5.

Listing 8-5: It's Warm and Cozy in Here

```
import java.util.Scanner;

class CelsiusToFahrenheit {

  public static void main(String args[]) {

    Scanner myScanner = new Scanner(System.in);
    double celsius, fahrenheit;
```

```
        System.out.print("Enter the Celsius temperature: ");
        celsius = myScanner.nextDouble();

        fahrenheit = 9.0 / 5.0 * celsius + 32.0;

        System.out.print("Room temperature? ");
        System.out.println(fahrenheit == 69.8);
    }
}
```

If you run the code in Listing 8-5 and input the number 21, the computer finds the value of `9.0 / 5.0 * 21 + 32.0`. Believe it or not, you want to check the computer's answer. (Who knows? Maybe the computer gets it wrong!) You need to do some arithmetic, but please don't reach for your calculator. A calculator is just a small computer, and machines of that kind stick up for one another. To check the computer's work, you need to do the arithmetic by hand. What? You say you're math phobic? Well, don't worry. I've done all the math in Figure 8-11.

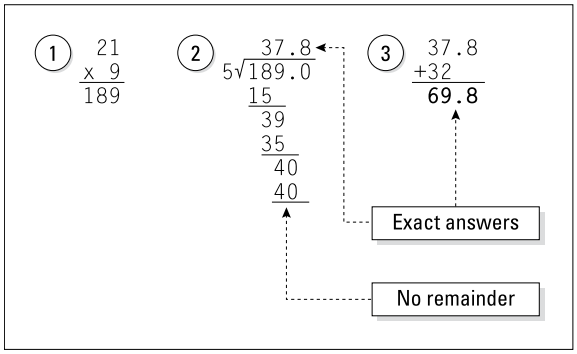


Figure 8-11: The Fahrenheit temperature is exactly 69.8.

If you do the arithmetic by hand, then the value you get for `9.0 / 5.0 * 21 + 32.0` is exactly 69.8. So run the code in Listing 8-5 and give `celsius` the value 21. You should get `true` when you display the value of `fahrenheit == 69.8`, right?

Well, no. Take a look at the run in Figure 8-12. When the computer evaluates `fahrenheit == 69.8`, the value turns out to be `false`, not `true`. What's going on here?

```
Enter the Celsius temperature: 21
Room temperature? false
```

Figure 8-12: A run of the code in Listing 8-5.

A little detective work can go a long way. So review the facts:

- **Fact:** The value of `fahrenheit` should be exactly 69.8.
- **Fact:** If `fahrenheit` is 69.8, then `fahrenheit == 69.8` is `true`.
- **Fact:** In Figure 8-12, the computer displays the word `false`. So the expression `fahrenheit == 69.8` isn't `true`.

How do you reconcile these facts? There can be little doubt that `fahrenheit == 69.8` is `false`, so what does that say about the value of `fahrenheit`? Nowhere in Listing 8-5 is the value of `fahrenheit` displayed. Could that be the problem?

At this point, I use a popular programmer's trick. I add statements to display the value of `fahrenheit`.

```
fahrenheit = 9.0 / 5.0 * celsius + 32.0;
System.out.print("fahrenheit: ");   //Added
System.out.println(fahrenheit);     //Added
```

A run of the enhanced code is shown in Figure 8-13. As you can see, the computer misses its mark. Instead of the expected value 69.8, the computer's value for `9.0 / 5.0 * 21 + 32.0` is 69.80000000000001. That's just the way the cookie crumbles. The computer does all its arithmetic with zeros and ones, so the computer's arithmetic doesn't look like the base-10 arithmetic in Figure 8-11. The computer's answer isn't wrong. The answer is just slightly inaccurate.

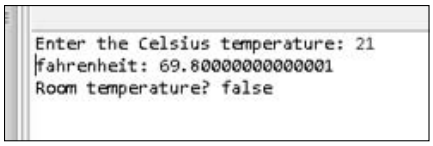


Figure 8-13: The fahrenheit variable's full value.

In an example in Chapter 7, Java's remainder operator (`%`) gives you the answer `0.1299999999999999` instead of the `0.13` that you expect. The same strange kind of thing happens in this section's example. But this section's code doesn't use an exotic remainder operator. This section's code uses your old friends — division, multiplication, and addition.

So be careful when you compare two numbers for equality (with `==`) or for inequality (with `!=`). Little inaccuracies can creep in almost anywhere when you work with Java's `double` type or with Java's `float` type. And several little inaccuracies can build on one another to become very large inaccuracies. When you compare two `double` values or two `float` values, the values are almost never dead-on equal to one another.

If your program isn't doing what you think it should do, then check your suspicions about the values of variables. Add `print` and `println` statements to your code.

When you compare `double` values, give yourself some leeway. Instead of comparing for exact equality, ask whether a particular value is reasonably close to the expected value. For example, use a condition like `fahrenheit >= 69.8 - 0.01 && fahrenheit <= 69.8 + 0.01` to find out whether `fahrenheit` is within 0.01 of the value 69.8. To read more about conditions containing Java's `&&` operator, see Chapter 10.

Automated debugging

If your program isn't working correctly, you can try something called a *debugger*. A debugger automatically adds invisible `print` and `println` calls to your suspicious code. In fact, debuggers have all kinds of features to help you diagnose problems. For example, a debugger can pause a run of your program and accept special commands to display variables' values. With some debuggers, you can pause a run and change a variable's value (just to see if things go better when you do).

An Eclipse *perspective* is a collection of views intended to help you with a certain aspect of program development. By default, Eclipse starts in the *Java perspective* — the arrangement of views to help you create Java programs. Another perspective — the *Debug perspective* — helps you diagnose errors in your code.

In this book, I don't promote the use of an automated debugger. But for any large programming project, automated debugging is an essential tool. So if you plan to write bigger and better programs, please give Eclipse's Debug perspective a try. For a small sample of the Debug perspective's capabilities, do the following:

1. **In the editor (where you see your Java code) double-click in the margin to the left of a line of code.**

   A little blue dot appears in the margin (see figure). This dot indicates a *breakpoint* in the code. In the steps that follow, you'll make the run of the program pause at this breakpoint. In the figure, I click the last line of code from Listing 8-5.

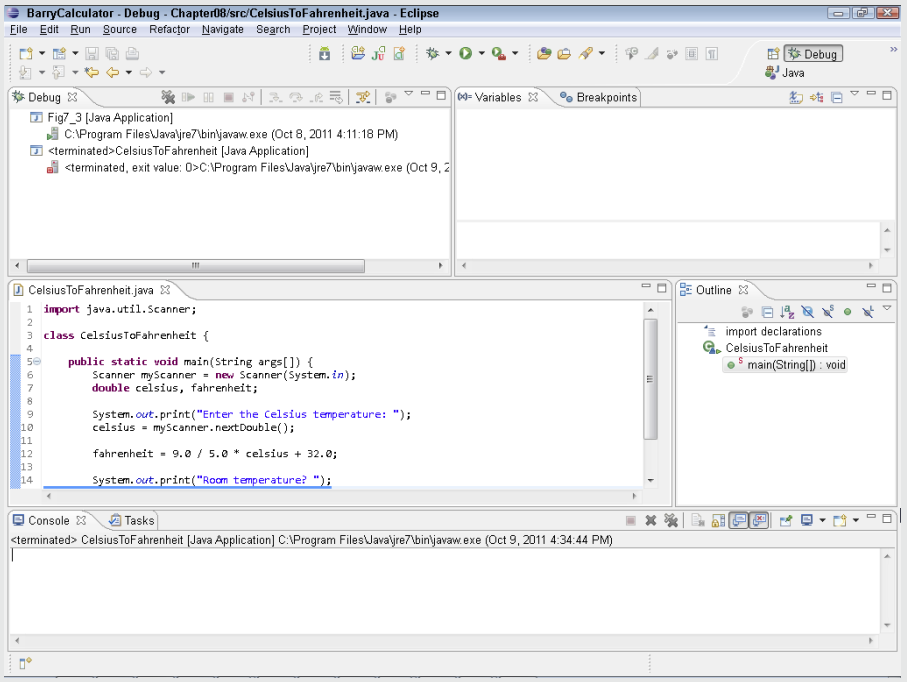
```
14          System.out.print("Room temperature? ");
15          System.out.println(fahrenheit == 69.8);
16      }
17  }
18
```

(continued)

(continued)

2. **In Eclipse's main menu, click Window➪Open Perspective➪Debug.**

   As a result, Eclipse displays a new layout. The new layout contains some familiar views, such as the Console view and the Outline view. The layout also contains some new views, such as the Debug view, the Variables view, and the Breakpoints view (see figure).



3. **In Eclipse's main menu, click Run➪Debug As➪Java Application.**

   Remember to select the Debug As menu item. Selecting this item enables all the debugging tools.

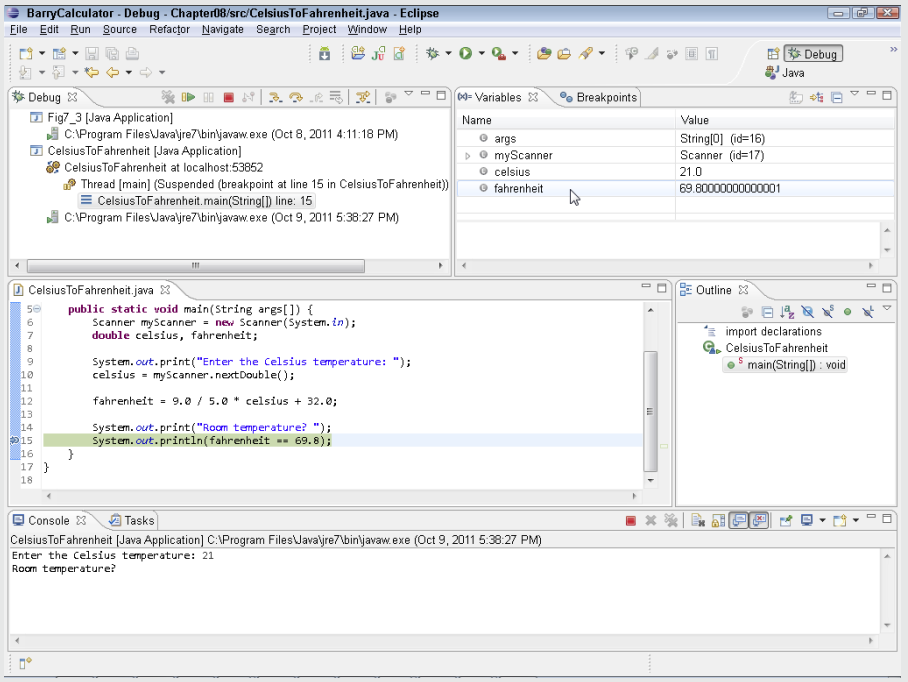
   Your code begins running. Because you're working with the program in Listing 8-5, the code prompts you to enter the Celsius temperature.

4. **Type the number `21` and then press Enter.**

   Your code continues running until execution reaches the breakpoint. At the breakpoint, the execution pauses to allow you to examine the program's state.

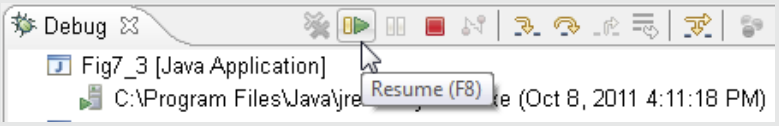
5. **In the upper-right corner of Eclipse's window, look for the Variables view.**

   The Variables view displays the values of the program's variables. (That's not surprising.) In the figure, the `fahrenheit` variable's value is 69.80000000000001. How nice! Using the debugging tools, you can examine variables' values in the middle of a run!



6. **To finish running your program, click the Resume button at the top of the Debug view (see figure).**

7. **To return to the Java perspective, click Window➪Open Perspective➪Java.**



Comparing characters

The comparison operators in Table 8-1 work overtime for characters. Roughly speaking, the operator < means "comes earlier in the alphabet." But you have to be careful of the following:

- Because `B` comes alphabetically before `H`, the condition `'B' < 'H'` is true. That's not surprising.
- Because `b` comes alphabetically before `h`, the condition `'b' < 'h'` is true. That's no surprise, either.
- Every uppercase letter comes before any of the lowercase letters, so the condition `'b' < 'H'` is *false*. Now that's a surprise (see Figure 8-14).

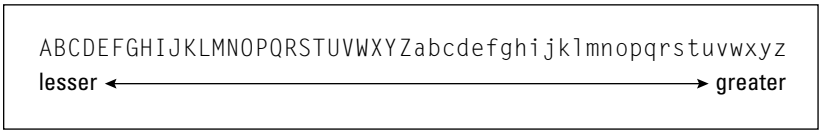


Figure 8-14: The ordering of the letters.

In practice, you seldom have reason to compare one letter with another. But in Chapter 18, you can read about Java's `String` type. With the `String` type, you can compare words, names, and other good stuff. At that point, you have to think carefully about alphabetical ordering, and the ideas in Figure 8-14 come in handy.

Under the hood, the letters `A` through `Z` are stored with numeric codes 65 through 90. The letters `a` through `z` are stored with codes 97 through 122. That's why each uppercase letter is "less than" any of the lowercase letters.

The Remaining Primitive Types

In Chapter 7, I tell you that Java has eight primitive types, but Table 7-1 lists only six out of eight types. Table 8-2 describes the remaining two types — the types `char` and `boolean`. Table 8-2 isn't too exciting, but I can't just leave you with the incomplete story in Table 7-1.

Table 8-2 Java's Primitive Non-numeric Types

Type Name	*Range of Values*
Character Type	
`char`	Thousands of characters, glyphs, and symbols
Logical Type	
`boolean`	Only `true` or `false`

If you dissect parts of the Java virtual machine, you find that Java considers `char` to be a numeric type. That's because Java represents characters with something called *Unicode* — an international standard for representing alphabets of the world's many languages. For example, the Unicode representation of an uppercase letter C is 67. The representation of a Hebrew letter aleph is 1488. And (to take a more obscure example) the representation for the voiced

retroflex approximant in phonetics is 635. But don't worry about all this. The only reason I'm writing about the `char` type's being numeric is to save face among my techie friends.

After looking at Table 8-2, you may be wondering what a glyph is. (In fact, I'm proud to be writing about this esoteric concept, whether you have any use for the information or not.) A *glyph* is a particular representation of a character. For example, a and *a* are two different glyphs, but both of these glyphs represent the same lowercase letter of the Roman alphabet. (Because these two glyphs have the same meaning, the glyphs are called *allographs*. If you want to sound smart, find a way to inject the words *glyph* and *allograph* into a casual conversation!)

Part III
Controlling the Flow

In this part . . .

A computer program is like a role-playing video game. It's not the kind of game that involves shooting, punching, or racing. It's a game that involves strategies. Find the golden ring to open the secret passageway. Save the princess by reciting the magic words. It's that sort of thing.

So in this part of the book, you create passageways. As your program weaves its way from one virtual room to another, the computer gets closer and closer to the solution of an important problem.

Hey, admit it. This sounds like fun!

Chapter 9

Forks in the Road

In This Chapter

- Writing statements that choose between alternatives
- Putting statements inside one another
- Writing several kinds of decision-making statements

Here's an excerpt from *Beginning Programming with Java For Dummies*, 3rd Edition, Chapter 2:

> **If you're trying to store words or sentences (not just single letters),** then you need to use something called a *String*. *

The excerpt illustrates two important points: First, you may have to use something called a *String*. Second, your choice of action can depend on something's being true or false.

> **If it's true** that you're trying to store words or sentences,
>
> you need to use something called a *String*.

This chapter deals with decision-making, which plays a fundamental role in the creation of instructions. With the material in this chapter, you expand your programming power by leaps and bounds.

Decisions, Decisions!

Picture yourself walking along a quiet country road. You're enjoying a pleasant summer day. It's not too hot, and a gentle breeze from the north makes you feel fresh and alert. You're holding a copy of this book, opened to Chapter 9. You read the paragraph about storing words or sentences, and then you look up.

---

* *This excerpt is reprinted with permission from John Wiley & Sons, Inc. If you can't find a copy of Beginning Programming with Java For Dummies,* 3rd Edition *in your local bookstore, visit* http://www.wiley.com.

You see a fork in the road. You see two signs — one pointing to the right; the other pointing to the left. One sign reads, "Storing words or sentences? True." The other sign reads, "Storing words or sentences? False." You evaluate the words-or-sentences situation and march on, veering right or left depending on your software situation. A diagram of this story is shown in Figure 9-1.

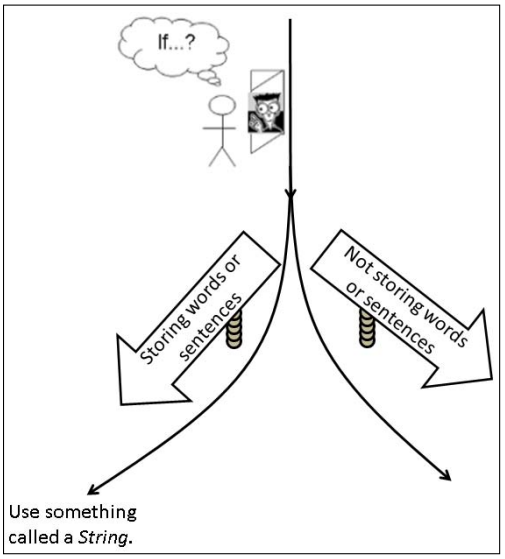


Figure 9-1: Which way to go?

Life is filled with forks in the road. Take an ordinary set of directions for heating up a frozen snack:

- **Microwave cooking directions:**
  - Place on microwave-safe plate.
  - Microwave on high for 2 minutes.
  - Turn product.
  - Microwave on high for 2 more minutes.
- **Conventional oven directions:**
  - Preheat oven to 350 degrees.
  - Place product on baking sheet.
  - Bake for 25 minutes.

Again, you choose between alternatives. If you use a microwave oven, do this. Otherwise, do that.

In fact, it's hard to imagine useful instructions that don't involve choices. If you're a homeowner with two dependents earning more than $30,000 per year, check here. If you don't remember how to use curly braces in Java programs, see Chapter 4. Did the user correctly type his password? If yes, then let the user log in; if no, then kick the bum out. If you think the market will go up, then buy stocks; otherwise, buy bonds. And if you buy stocks, which should you buy? And when should you sell?

Making Decisions (Java if Statements)

When you work with computer programs, you make one decision after another. Almost every programming language has a way of branching in one of two directions. In Java (and in many other languages), the branching feature is called an *if statement*. Check out Listing 9-1 to see an `if` statement.

Listing 9-1: An if Statement

```
if (randomNumber > 5) {
    System.out.println("Yes. Isn't it obvious?");
} else {
    System.out.println("No, and don't ask again.");
}
```

To see a complete program containing the code from Listing 9-1, skip to Listing 9-2 (or, if you prefer, walk, jump, or run to Listing 9-2).

The `if` statement in Listing 9-1 represents a branch, a decision, two alternative courses of action. In plain English, this statement has the following meaning:

```
If the randomNumber variable's value is greater than 5,
    display "Yes. Isn't it obvious?" on the screen.
Otherwise,
    display "No, and don't ask again." on the screen.
```

Pictorially, you get the fork shown in Figure 9-2.

Looking carefully at if statements

An `if` statement can take the following form:

```
if (Condition) {
    SomeStatements
} else {
    OtherStatements
}
```

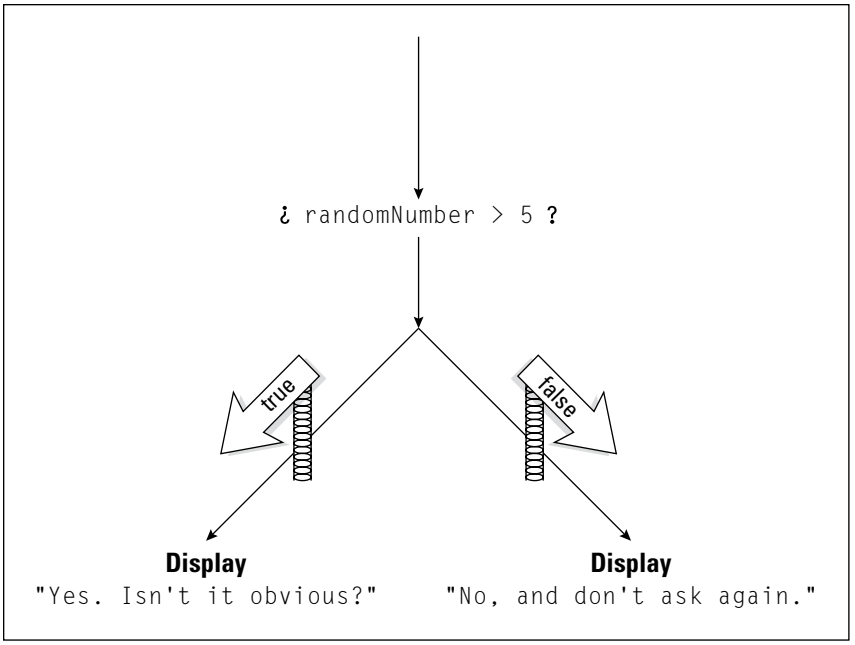


Figure 9-2: A random number decides your fate.

To get a real-life `if` statement, substitute meaningful text for the three placeholders *`Condition`*, *`SomeStatements`*, and *`OtherStatements`*. Here's how I make the substitutions in Listing 9-1:

- I substitute `randomNumber > 5` for *`Condition`*.
- I substitute `System.out.println("Yes. Isn't it obvious?");` for *`SomeStatements`*.
- I substitute `System.out.println("No, and don't ask again.");` for *`OtherStatements`*.

The substitutions are illustrated in Figure 9-3.

Sometimes I need alternate names for parts of an `if` statement. I call them the *if clause* and the *else clause*.

```
if (Condition) {
    if clause
} else {
    else clause
}
```

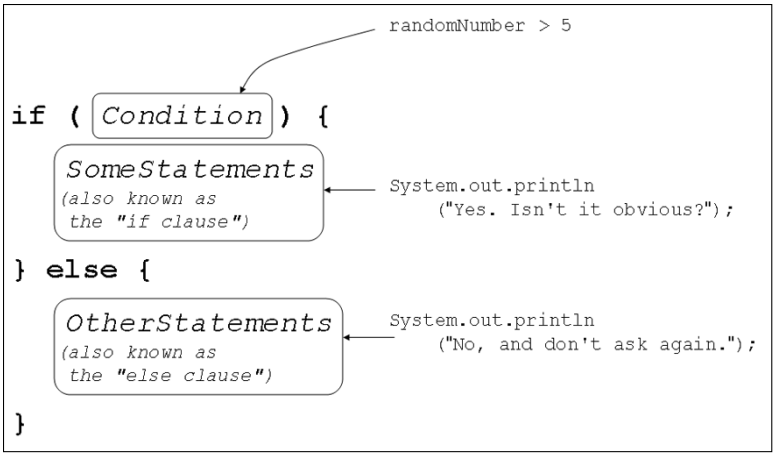


Figure 9-3: An if statement and its format.

An `if` statement is an example of a *compound statement* — a statement that includes other statements within it. The `if` statement in Listing 9-1 includes two `println` calls, and these calls to `println` are statements.

Notice how I use parentheses and semicolons in the `if` statement of Listing 9-1. In particular, notice the following:

- The condition must be in parentheses.
- Statements inside the `if` clause end with semicolons. So do statements inside the `else` clause.
- There's no semicolon immediately after the condition.
- There's no semicolon immediately after the word `else`.

As a beginning programmer, you may think these rules are arbitrary. But they're not. These rules belong to a very carefully crafted grammar. They're like the grammar rules for English sentences, but they're even more logical! (Sorry, Kelly.)

Table 9-1 shows you the kinds of things that can go wrong when you break the `if` statement's punctuation rules. The table's last two items are the most notorious. In these two situations, the compiler doesn't catch the error. This lulls you into a false sense of security. The trouble is, when you run the program, the code's behavior isn't what you expect it to be.

Table 9-1 Common if Statement Error Messages

Error	*Example*	*Most Likely Messages or Results*
Missing parentheses surrounding the condition	`if randomNumber > 5 {`	'(' expected Syntax error on token "if", (expected after this token

Missing semicolon after a statement that's inside the if clause or the else clause	`if (randomNumber > 5) {`
	`System.out.println("Y")`
	`}`
		';' expected
		
		Syntax error, insert ";" to complete BlockStatements

Semicolon immediately after the condition	`if (randomNumber > 5); {`
	`System.out.println("Y");`
	`} else {`
		'else' without 'if'
		
		Syntax error on token "else", delete this token

Semicolon immediately after the word else	`} else; {`	The program compiles without errors, but the statement after the word else is always executed, whether the condition is true or false.

Missing curly braces	`if (randomNumber > 5)`
	`System.out.println("Y");`
	`else`
	`System.out.println("N");`
		The program sometimes compiles without errors, but the program's run may not do what you expect it to do. (So the bottom line is, don't omit the curly braces.)

As you compose your code, it helps to think of an `if` statement as one indivisible unit. Instead of typing the whole first line (condition and all), try typing the `if` statement's skeletal outline.

```
if () {           //To do: Fill in the condition.
                  //To do: Fill in SomeStatements.
} else {
                  //To do: Fill in OtherStatements.
}
```

With the entire outline in place, you can start working on the items on your to-do list. When you apply this kind of thinking to a compound statement, it's harder to make a mistake.

A complete program

Listing 9-2 contains a complete program with a simple `if` statement. The listing's code behaves like an electronic oracle. Ask the program a yes or no question, and the program answers you back. Of course, the answer to your question is randomly generated. But who cares? It's fun to ask anyway.

Listing 9-2: I Know Everything

```
import java.util.Scanner;
import java.util.Random;

class AnswerYesOrNo {

  public static void main(String args[]) {
    Scanner myScanner = new Scanner(System.in);
    Random myRandom = new Random();
    int randomNumber;

    System.out.print("Type your question, my child:  ");
    myScanner.nextLine();

    randomNumber = myRandom.nextInt(10) + 1;

    if (randomNumber > 5) {
      System.out.println("Yes. Isn't it obvious?");
    } else {
      System.out.println("No, and don't ask again.");
    }
  }
}
```

Figure 9-4 shows several runs of the program in Listing 9-2. The program's action has four parts:

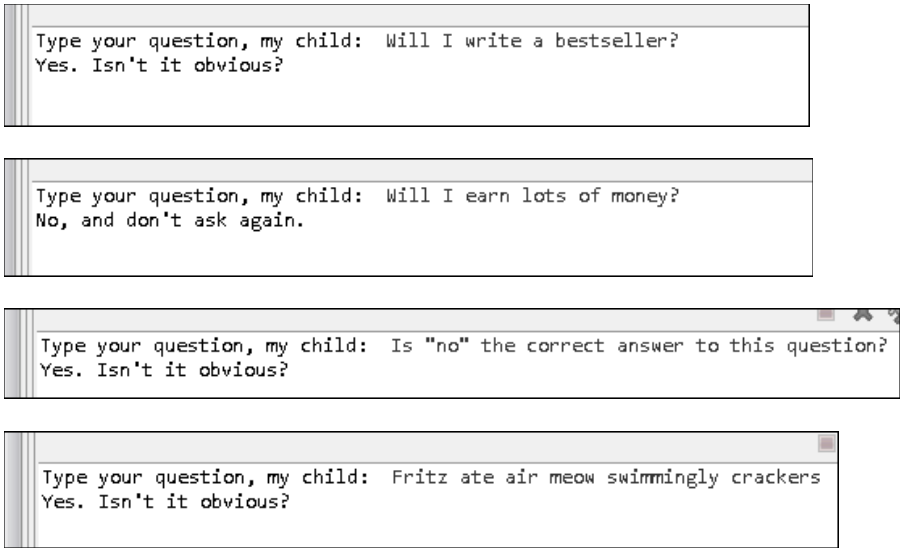

Type your question, my child: Will I write a bestseller?
Yes. Isn't it obvious?
Type your question, my child: Will I earn lots of money?
No, and don't ask again.
Type your question, my child: Is "no" the correct answer to this question?
Yes. Isn't it obvious?
Type your question, my child: Fritz ate air meow swimmingly crackers
Yes. Isn't it obvious?

Figure 9-4: The all-knowing Java program in action.

1. **Prompt the user.**

   Call `System.out.print`, telling the user to type a question.

2. **Get the user's question from the keyboard.**

   In Figure 9-4, I run the `AnswerYesOrNo` program four times, and I type a different question each time. Meanwhile, back in Listing 9-2, the statement

   ```
   myScanner.nextLine();
   ```

   swallows up my question and does absolutely nothing with it. This is an anomaly, but you're smart, so you can handle it.

   Normally, when a program gets input from the keyboard, the program does something with the input. For example, the program can assign the input to a variable:

   ```
   amount = myScanner.nextDouble();
   ```

   Alternatively, the program can display the input on the screen:

   ```
   System.out.println(myScanner.nextLine());
   ```

   But the code in Listing 9-2 is different. When this `AnswerYesOrNo` program runs, the user has to type something. (The call to `getLine` waits for the user to type some stuff and then press Enter.) But the `AnswerYesOrNo` program has no need to store the input for further analysis. (The computer does what I do when my wife asks me if I plan to clean up after myself. I ignore the question and make up an arbitrary answer.) So the program doesn't do anything with the user's input. The call to `myScanner.nextLine` just sits there in a statement of its own,

doing nothing, behaving like a big black hole. It's unusual for a program to do this, but an electronic oracle is an unusual thing. It calls for some slightly unusual code.

3. **Get a random number — any `int` value from 1 to 10.**

   Okay, wise guys. You've just trashed the user's input. How will you answer yes or no to the user's question?

   No problem! None at all! You'll display an answer randomly. The user won't know the difference. (Hah, hah!) You can do this as long as you can generate random numbers. The numbers from 1 to 10 will do just fine.

   In Listing 9-2, the stuff about `Random` and `myRandom` looks very much like the familiar `Scanner` code. From a beginning programmer's point of view, `Random` and `Scanner` work almost the same way. Of course, there's an important difference. A call to the `Random` class's `nextInt(10)` method doesn't fetch anything from the keyboard. Instead, this `nextInt(10)` method gets a number out of the blue.

   The name `Random` is defined in the Java API. The call to `myRandom.nextInt(10)` in Listing 9-2 gets a number from 0 to 9. Then my code adds 1 (making a number from 1 to 10) and assigns that number to the variable `randomNumber`. When that's done, you're ready to answer the user's question.

   In Java's API, the word `Random` is the name of a Java class, and `nextInt` is the name of a Java method. For more information on the relationship between classes and methods, see Chapters 17, 18, and 19.

4. **Answer yes or no.**

   Calling `myRandom.nextInt(10)` is like spinning a wheel on a TV game show. The wheel has slots numbered 1 to 10. The `if` statement in Listing 9-2 turns your number into a yes or no alternative. If you roll a number that's greater than 5, the program answers *yes.* Otherwise (if you roll a number that's less than or equal to 5), the program answers *no.*

   You can trust me on this one. I've made lots of important decisions based on my `AnswerYesOrNo` program.

Indenting if statements in your code

Notice how, in Listing 9-2, the `println` calls inside the `if` statement are indented. Strictly speaking, you don't have to indent the statements that are inside an `if` statement. For all the compiler cares, you can write your whole program on a single line or place all your statements in an artful, misshapen zigzag. The problem is, if you don't indent your statements in some logical fashion, then neither you nor anyone else can make sense of your code. In Listing 9-2, the indenting of the `println` calls helps your eye (and brain) see quickly that these statements are subordinate to the overall `if/else` flow.

Randomness makes me dizzy

When you call `myRandom.nextInt(10) + 1`, you get a number from 1 to 10. As a test, I wrote a program that calls the `myRandom.nextInt(10) + 1` 20 times.

```
Random myRandom=new Random();
System.out.print
  (myRandom.nextInt(10) + 1);
System.out.print(" ");
System.out.print
  (myRandom.nextInt(10) + 1);
System.out.print(" ");
System.out.print
  (myRandom.nextInt(10) + 1);
//...And so on.
```

I ran the program several times, and got the results shown in the following figure. (Actually, I copied the results from Eclipse's Console view to Windows Notepad.) Stare briefly at the figure and notice two trends:

- There's no obvious way to predict what number comes next.
- No number occurs much more often than any of the others.

The Java virtual machine jumps through hoops to maintain these trends. That's because cranking out numbers in a random fashion is a very tricky business. Here are some interesting facts about the process:

- Scientists and nonscientists use the term *random number.* But in reality, there's no such thing as a single random number. After all, how random is a number like 9?

  A number is *random* only when it's one in a very disorderly collection of numbers. More precisely, a number is *random* if the process used to generate the number follows the two trends listed above. When they're being careful, scientists avoid the term *random number,* and use the term *randomly generated number* instead.

- It's hard to generate numbers randomly. Computer programs do the best they can, but ultimately, today's computer programs follow a pattern, and that pattern isn't truly random.

  To generate numbers in a truly random fashion, you need a big tub of ping-pong balls, like the kind they use in state lottery drawings. The problem is, most computers don't come with big tubs of ping-pong balls among their peripherals. So strictly speaking, the numbers generated by Java's `Random` class aren't random. Instead, scientists call these numbers *pseudorandom*.

- It surprises us all, but knowing one randomly generated value is of no help in predicting the next randomly generated value.

  For example, if you toss a coin twice, and get heads each time, are you more likely to get tails on the third flip? No. It's still 50-50.

  If you have three sons, and you're expecting a fourth child, is the fourth child more likely to be a girl? No. A child's gender has nothing to do with the genders of the older children. (I'm ignoring any biological effects, which I know absolutely nothing about. Wait! I do know some biological trivia: A newborn child is more likely to be a boy than a girl. For every 21 newborn boys, there are only 20 newborn girls. Boys are weaker, so we die off faster. That's why nature makes more of us at birth.)

In a small program, unindented or poorly indented code is barely tolerable. But in a complicated program, indentation that doesn't follow a neat, logical pattern is a big, ugly nightmare.

Always indent your code to make the program's flow apparent at a glance.

Variations on the Theme

I don't like to skin cats. But I've heard that, if I ever need to skin one, I have a choice of several techniques. I'll keep that in mind the next time my cat Histamine mistakes the carpet for a litter box.*

Anyway, whether you're skinning catfish, skinning kitties, or writing computer programs, the same principle holds true. You always have alternatives. Listing 9-2 shows you one way to write an `if` statement. The rest of this chapter (and all of Chapter 10) show you some other ways to create `if` statements.

. . . Or else what?

You can create an `if` statement without an `else` clause. For example, imagine a web page on which one in ten randomly chosen visitors receives a special offer. To keep visitors guessing, I call the `Random` class's `nextInt` method, and make the offer to anyone whose number is lucky 7.

- If `myRandom.nextInt(10) + 1` generates the number 7, display a special offer message.
- If `myRandom.nextInt(10) + 1` generates any number other than 7, do nothing. Don't display a special offer message and don't display a discouraging, "Sorry, no offer for you," message.

The code to implement such a strategy is shown in Listing 9-3. A few runs of the code are shown in Figure 9-5.

*Rick Ross, who read about skinning cats in one of my other books, sent me this information via e-mail: ". . . on page 10 you refer to 'skinning the cat' and go on to discuss litter boxes and whatnot. Please note that the phrase 'more than one way to skin a cat' refers to the difficulty in removing the inedible skin from catfish, and that there is more than one way to do same. These range from nailing the critter's tail to a board and taking a pair of pliers to peel it down, to letting the furry kind of cat have the darn thing and just not worrying about it. I grew up on The River (the big one running north/south down the US that begins with 'M' and has so many repeated letters), so it's integral to our experience there. A common misconception (if inhumane and revolting). Just thought you'd want to know."

Listing 9-3: Aren't You Lucky?

```
import java.util.Random;

class SpecialOffer {

    public static void main(String args[]) {
        Random myRandom = new Random();
        int randomNumber = myRandom.nextInt(10) + 1;

        if (randomNumber == 7) {
            System.out.println("An offer just for you!");
        }

        System.out.println(randomNumber);
    }
}
```

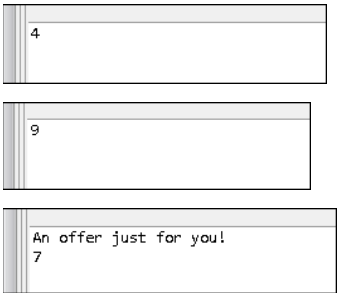


Figure 9-5: Three runs of the code in Listing 9-3.

The `if` statement in Listing 9-3 has no `else` clause. This `if` statement has the following form:

```
if (Condition) {
    SomeStatements
}
```

When `randomNumber` is 7, the computer displays `An offer just for you!` When `randomNumber` isn't 7, the computer doesn't display `An offer just for you!` The action is illustrated in Figure 9-6.

Always (I mean *always*) use a double equal sign when you compare two numbers or characters in an `if` statement's condition. Never (that's *never, ever, ever*) use a single equal sign to compare two values. A single equal sign does assignment, not comparison.

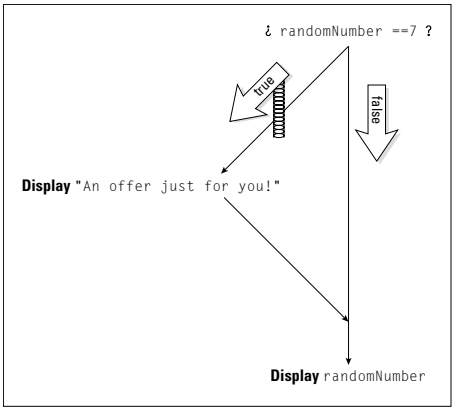


Figure 9-6: If you have nothing good to say, then don't say anything.

In Listing 9-3, I took the liberty of adding an extra `println`. This `println` (at the end of the `main` method) displays the random number generated by my call to `nextInt`. On a webpage with special offers, you probably wouldn't see the randomly generated number, but I can't test my `SpecialOffer` code without knowing what numbers the code generates.

Anyway, notice that the value of `randomNumber` is displayed in every run. The `println` for `randomNumber` isn't inside the `if` statement. (This `println` comes after the `if` statement.) So the computer always executes this `println`. Whether `randomNumber == 7` is true or false, the computer takes the appropriate `if` action, and then marches on to execute `System.out.println(randomNumber)`.

Packing more stuff into an if statement

Here's an interesting situation: You have two baseball teams — the Hankees and the Socks. You want to display the teams' scores on two separate lines, with the winner's score coming first. (On the computer screen, the winner's score is displayed above the loser's score. In case of a tie, you display the two identical scores, one above the other.) Listing 9-4 has the code.

Listing 9-4: May the Best Team Be Displayed First

```
import java.util.Scanner;
import static java.lang.System.in;
import static java.lang.System.out;

class TwoTeams {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(in);
```

(continued)

Listing 9-4 *(continued)*

```
        int hankees, socks;

        out.print("Hankees and Socks scores?  ");
        hankees = myScanner.nextInt();
        socks = myScanner.nextInt();
        out.println();

        if (hankees > socks) {
            out.print("Hankees: ");
            out.println(hankees);
            out.print("Socks:   ");
            out.println(socks);
        } else {
            out.print("Socks:   ");
            out.println(socks);
            out.print("Hankees: ");
            out.println(hankees);
        }
    }
}
```

Figure 9-7 has a few runs of the code. (To show a few runs in one figure, I copied the results from Eclipse's Console view to Windows Notepad.)

With curly braces, a bunch of `print` and `println` calls are tucked away safely inside the `if` clause. Another group of `print` and `println` calls are squished inside the `else` clause. This creates the forking situation shown in Figure 9-8.

```
Hankees and Socks scores?  9 4

Hankees: 9
Socks:   4



Hankees and Socks scores?  3 8

Socks:   8
Hankees: 3



Hankees and Socks scores?  0 0

Socks:   0
Hankees: 0
```

Figure 9-7: See? The code in Listing 9-4 really works!

Statements and blocks

An elegant way to think about `if` statements is to realize that you can put only one statement inside each clause of an `if` statement.

```
if (Condition)
    aStatement
else
    anotherStatement
```

On first reading of this one-statement rule, you're probably thinking that there's a misprint. After all, in Listing 9-4, each clause (the `if` clause and the `else` clause) seems to contain four statements, not just one.

But technically, the `if` clause in Listing 9-4 has only one statement, and the `else` clause in Listing 9-4 has only one statement. The trick is, when you surround a bunch of statements with curly braces, you get what's called a *block*, and a block behaves, in all respects, like a single statement. In fact, the official Java documentation lists a block as a kind of statement (one of many different kinds of statements). So in Listing 9-4, the block

```
{
    out.print("Hankees: ");
    out.println(hankees);
    out.print("Socks:   ");
    out.println(socks);
}
```

is a single statement. It's a statement that has four smaller statements within it. So this big block, this single statement, serves as the one and only statement inside the `if` clause in Listing 9-4.

That's how the one-statement rule works. In an `if` statement, when you want the computer to execute several statements, you combine those statements into one big statement. To do this, you make a block using curly braces.

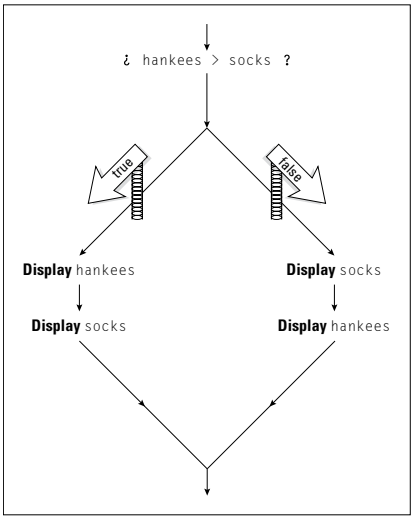


Figure 9-8: Cheer for your favorite team.

Some handy import declarations

When I wrote this section's example, I was tired of writing the word `System`. After all, Listing 9-4 has ten `System.out.print` lines. By this point in the book, shouldn't my computer remember what `out.print` means?

Of course, computers don't work that way. If you want a computer to "know" what `out.print` means, you have to code that knowledge somewhere inside the Java compiler.

Fortunately for me, the ability to abbreviate things like `System.out.print` is available from Java 5.0 onward. (An older Java compiler just chokes on the code in Listing 9-4.) This ability to abbreviate things is called *static import*. It's illustrated in the second and third lines of Listing 9-4.

Whenever I start a program with the line

```
import static java.lang.System.out;
```

I can replace `System.out` with plain `out` in the remainder of the program. The same holds true of `System.in`. With an import declaration near the top of Listing 9-4, I can replace `new Scanner(System.in)` with the simpler `new Scanner(in)`.

You may be wondering what all the fuss is about. If I can abbreviate `java.util.Scanner` by writing `Scanner`, what's so special about abbreviating `System.out`? And why do I have to write `out.print`? Can I trim `System.out.print` down to the single word `print`? Look again at the first few lines of Listing 9-4. When do I need the word `static`? And what's the difference between `java.util` and `java.lang`?

I'm sorry. My response to these questions won't thrill you. The fact is, I can't explain away any of these issues until Chapter 18. Before I can explain static import declarations, I need to introduce some ideas. I need to describe classes, packages, and static members.

So until you reach Chapter 18, please bear with me. Just paste three import declarations to the top of your Java programs and trust that everything will work well.

You can abbreviate `System.out` with the single word `out`. And you can abbreviate `System.in` with the single word `in`. Just be sure to copy the `import` declarations *exactly* as you see them in Listing 9-4. With any deviation from the lines in Listing 9-4, you may get a compiler error.

Chapter 10

Which Way Did He Go?

In This Chapter

- Untangling complicated conditions
- Writing cool conditional code
- Intertwining your `if` statements

It's tax time again. At the moment, I'm working on Form 12432-89B:

> If you're married with fewer than three children, and your income is higher than the EIQ (Estimated Income Quota), or if you're single and living in a non-residential area (as defined by Section 10, Part iii of the Uniform Zoning Act), and you're either self-employed as an LLC (Limited Liability Company) or you qualify for veterans benefits, then skip Steps 3 and 4 or 4, 5, and 6, depending on your answers to Questions 2a and 3d.

This chapter has nothing as complex as Form 12432-89B, but it does deal with the potential complexity of `if` statements.

Forming Bigger and Better Conditions

In Listing 9-2, the code chooses a course of action based on one call to the `Random` class's `nextInt` method. That's fine for the electronic oracle program described in Chapter 9, but what if you're rolling a pair of dice? In Backgammon and other dice games, rolling 3 and 5 isn't the same as rolling 4 and 4, even though the total for both rolls is 8. The next move varies, depending on whether or not you roll doubles. To get the computer to roll two dice, you execute `myRandom.nextInt(6) + 1` two times. Then you combine the two rolls into a larger, more complicated `if` statement.

So to simulate a Backgammon game (and many other, more practical situations) you need to combine conditions.

```
If die1 + die2 equals 8 and die1 equals die2, ...
```

You need things like *and* and *or* — things that can wire conditions together. Java has operators to represent these concepts, which are described in Table 10-1 and illustrated in Figure 10-1.

Table 10-1 Logical Operators

Operator Symbol	*Meaning*	*Example*	*Illustration*
`&&`	and	`4 < age && age < 8`	Figure 10-1(a)
`||`	or	`age < 4 || 8 < age`	Figure 10-1(b)
`!`	not	`!eachKidGetsTen`	Figure 10-1(c)

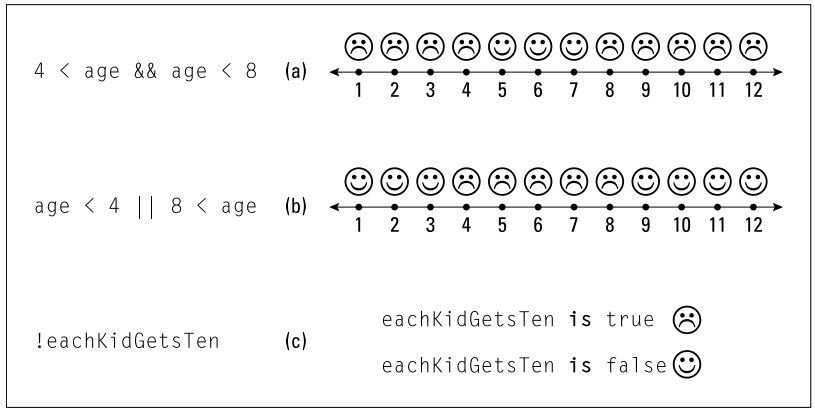

4 < age && age < 8
(a)
1 2 3 4 5 6 7 8 9 10 11 12
age < 4 || 8 < age
(b)
1 2 3 4 5 6 7 8 9 10 11 12
!eachKidGetsTen
(c)
eachKidGetsTen is true
eachKidGetsTen is false

Figure 10-1: When you satisfy a condition, you're happy.

Combined conditions, like the ones in Table 10-1, can be mighty confusing. That's why I tread carefully when I use such things. Here's a short explanation of each example in the table:

- `4 < age && age < 8`

  The value of the `age` variable is greater than 4 *and* is less than 8. The numbers 5, 6, 7, 8, 9 . . . are all greater than 4. But among these numbers, only 5, 6, and 7 are less than 8. So only the numbers 5, 6, and 7 satisfy this combined condition.

- `age < 4 || 8 < age`

  The value of the `age` variable is less than 4 *or* is greater than 8. To create the or condition, you use two pipe symbols. On many U.S. English keyboards, you can find the pipe symbol immediately above the Enter key (the same key as the backslash, but shifted).

In this combined condition, the value of the `age` variable is either less than 4 or is greater than 8. So for example, if a number is less than 4, then the number satisfies the condition. Numbers like 1, 2, and 3 are all less than 4, so these numbers satisfy the combined condition.

Also, if a number is greater than 8, then the number satisfies the combined condition. Numbers like 9, 10, and 11 are all greater than 8, so these numbers satisfy the condition.

- `!eachKidGetsTen`

  If I weren't experienced with computer programming languages, I'd be confused by the exclamation point. I'd think that `!eachKidGetsTen` means, "Yes, each kid *does* get ten." But that's not what this expression means. This expression says, "The variable `eachKidGetsTen` does *not* have the value `true`." In Java and other programming languages, an exclamation point stands for *negative,* for *no way,* for *not.*

  Listing 8-4 has a `boolean` variable named `eachKidGetsTen`. A `boolean` variable's value is either `true` or `false`. Because `!` means *not,* the expressions `eachKidGetsTen` and `!eachKidGetsTen` have opposite values. So when `eachKidGetsTen` is true, `!eachKidGetsTen` is false (and vice versa).

Java's `||` operator is *inclusive.* This means that you get `true` whenever the thing on the left side is `true`, the thing on the right side is `true`, or both things are `true`. For example, the condition `2 < 10 || 20 < 30` is true.

In Java, you can't combine comparisons the way you do in ordinary English. In English, you may say, "We'll have between three and ten people at the dinner table." But in Java, you get an error message if you write `3 <= people <= 10`. To do this comparison, you need to something like `3 <= people && people <= 10`.

Combining conditions: An example

Here's a handy example of the use of logical operators. A movie theater posts its prices for admission.

Regular price: $9.25

Kids under 12: $5.25

Seniors (65 and older): $5.25

Because the kids' and seniors' prices are the same, you can combine these prices into one category. (That's not always the best programming strategy, but do it anyway for this example.) To find a particular moviegoer's ticket price, you need one or more `if` statements. You can structure the conditions in many ways, and I chose one of these ways for the code in Listing 10-1.

Listing 10-1: Are You Paying Too Much?

```
import java.util.Scanner;

class TicketPrice {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        int age;
        double price = 0.00;

        System.out.print("How old are you? ");
        age = myScanner.nextInt();

        if (age >= 12 && age < 65) {
            price = 9.25;
        }
        if (age < 12 || age >= 65) {
            price = 5.25;
        }

        System.out.print("Please pay $");
        System.out.print(price);
        System.out.print(". ");
        System.out.println("Enjoy the show!");
    }
}
```

Several runs of the `TicketPrice` program (see Listing 10-1) are shown in Figure 10-2. (For your viewing pleasure, I've copied the runs from Eclipse's Console view to Windows Notepad.) When you turn 12, you start paying full price. You keep paying full price until you become 65. At that point, you pay the reduced price again.

The pivotal part of Listing 10-1 is the lump of `if` statements in the middle, which are illustrated in Figure 10-3.

```
How old are you? 11
Please pay $5.25. Enjoy the show!

How old are you? 12
Please pay $9.25. Enjoy the show!

How old are you? 35
Please pay $9.25. Enjoy the show!

How old are you? 64
Please pay $9.25. Enjoy the show!

How old are you? 65
Please pay $5.25. Enjoy the show!
```

Figure 10-2: Admission prices for *Beginning Programming with Java For Dummies: The Movie.*

- The first `if` statement's condition tests for the regular price group. Anyone who's at least 12 years of age *and* is under 65 belongs in this group.
- The second `if` statement's condition tests for the fringe ages. A person who's under 12 *or* is 65 or older belongs in this category.

When you form the opposite of an existing condition, you can often follow the pattern in Listing 10-1. The opposite of `>=` is `<`. The opposite of `<` is `>=`. The opposite of `&&` is `||`.

If you change the dollar amounts in Listing 10-1, you can get into trouble. For example, with the statement `price = 5.00`, the program displays `Please pay $5.0. Enjoy the show!` This happens because Java doesn't store the two zeros to the right of the decimal point (and Java doesn't know or care that 5.00 is a dollar amount). To fix this kind of thing, see the discussion of `NumberFormat.getCurrencyInstance` in Chapter 18.

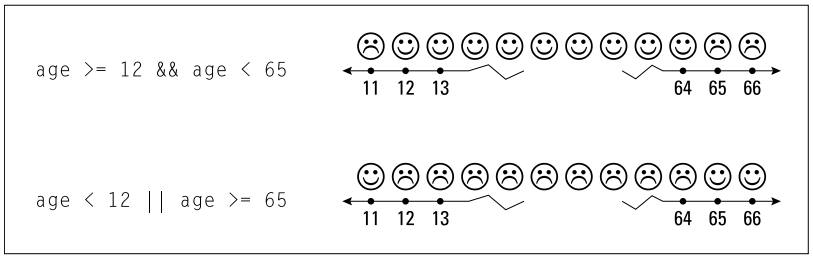


Figure 10-3: The meanings of the conditions in Listing 10-1.

When to initialize?

Take a look at Listing 10-1 and notice the `price` variable's initialization:

```
double price = 0.00;
```

This line declares the `price` variable and sets the variable's starting value to `0.00`. When I omit this initialization, I get an error message:

```
The local variable price may not have been initialized
```

What's the deal here? I don't initialize the `age` variable, but the compiler doesn't complain about that. Why is the compiler fussing over the `price` variable? The answer is in the placement of the code's assignment statements. Consider the following two facts:

- **The statement that assigns a value to `age` (`age = myScanner.nextInt()`) is not inside an `if` statement.**

  That assignment statement always gets executed, and (as long as nothing extraordinary happens) the variable `age` is sure to be assigned a value.

- **Both statements that assign a value to `price` (`price = 9.25` and `price = 5.25`) are inside `if` statements.**

  If you look at Figure 10-3, you see that every age group is covered. No one shows up at the ticket counter with an age that forces both `if` conditions to be `false`. So whenever you run the `TicketPrice` program, either the first or the second `price` assignment is executed.

  The problem is that the compiler isn't smart enough to check all this. The compiler just sees the structure in Figure 10-4 and becomes scared that the computer won't take either of the `true` detours.

  If (for some unforeseen reason) both of the `if` statements' conditions are `false`, then the variable `price` doesn't get assigned a value. So without an initialization, `price` has no value. (More precisely, `price` has no value that's intentionally given to it in the code.)

  Eventually, the computer reaches the `System.out.print(price)` statement. It can't display `price` unless `price` has a meaningful value. So at that point, the compiler throws up its virtual hands in disgust.

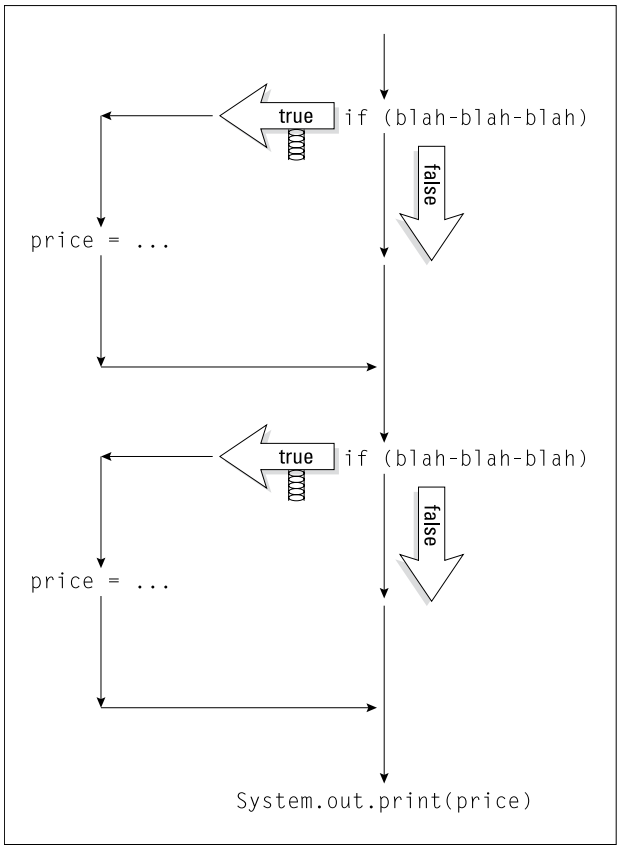


Figure 10-4: The choices in Listing 10-1.

More and more conditions

Last night I got a discount coupon along with the meal at the local burger joint. The coupon is good for $2.00 off a ticket at the local movie theater. To make use of the coupon in the `TicketPrice` program, I have to tweak the code in Listing 10-1. The revised code is in Listing 10-2. In Figure 10-5, I take that new code around the block a few times.

Listing 10-2: Do You Have a Coupon?

```
import java.util.Scanner;

class TicketPriceWithDiscount {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        int age;
        double price = 0.00;
        char reply;

        System.out.print("How old are you? ");
        age = myScanner.nextInt();

        System.out.print("Have a coupon? (Y/N) ");
        reply = myScanner.findWithinHorizon(".", 0)
                                              .charAt(0);

        if (age >= 12 && age < 65) {
            price = 9.25;
        }
        if (age < 12 || age >= 65) {
            price = 5.25;
        }

        if (reply == 'Y' || reply == 'y') {
            price -= 2.00;
        }
        if (reply != 'Y' && reply != 'y' &&
            reply!='N' && reply!='n') {
            System.out.println("Huh?");
        }

        System.out.print("Please pay $");
        System.out.print(price);
        System.out.print(". ");
        System.out.println("Enjoy the show!");
    }
}
```

```
How old are you? 51
Have a coupon? (Y/N) Y
Please pay $7.25. Enjoy the show!

How old are you? 51
Have a coupon? (Y/N) y
Please pay $7.25. Enjoy the show!

How old are you? 51
Have a coupon? (Y/N) N
Please pay $9.25. Enjoy the show!

How old are you? 51
Have a coupon? (Y/N) X
Huh?
Please pay $9.25. Enjoy the show!
```

Figure 10-5: Running the code in Listing 10-2.

Listing 10-2 has two `if` statements whose conditions involve characters:

- In the first such statement, the computer checks to see whether the `reply` variable stores the letter `Y` *or* the letter `y`. If either is the case, then it subtracts 2.00 from the `price`. (For information on operators like `-=`, see Chapter 7.)
- The second such statement has a hefty condition. The condition tests to see whether the `reply` variable stores any reasonable value at all. If the reply *isn't* `Y`, *and isn't* `y`, *and isn't* `N`, *and isn't* `n`, then the computer expresses its concern by displaying, "Huh?" (As a paying customer, the word "Huh?" on the automated ticket teller's screen will certainly get your attention.)

When you create a big multipart condition, you always have several ways to think about the condition. For example, you can rewrite the last condition in Listing 10-2 as `if (!(reply == 'Y' || reply == 'y' || reply == 'N' || reply == 'n'))`. "*If it's not the case that* the `reply` is *either* `Y`, `y`, `N`, *or* `n`, then display 'Huh?'" So which way of writing the condition is better — the way I do it in Listing 10-2, or the way I do it in this tip? It depends on your taste. Whatever makes the logic easiest for you to understand is the best way.

Using boolean variables

No matter how good a program is, you can always make it a little bit better. Take the code in Listing 10-2. Does the forest of `if` statements make you nervous? Do you slow to a crawl when you read each condition? Wouldn't it be nice if you could glance at a condition and make sense of it very quickly?

To some extent, you can. If you're willing to create some additional variables, you can make your code easier to read. Listing 10-3 shows you how.

Listing 10-3: George Boole Would Be Proud

```
import java.util.Scanner;

class NicePrice {
```

```
    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        int age;
        double price = 0.00;
        char reply;
        boolean isKid, isSenior, hasCoupon, hasNoCoupon;

        System.out.print("How old are you? ");
        age = myScanner.nextInt();

        System.out.print("Have a coupon? (Y/N) ");
        reply = myScanner.findWithinHorizon(".", 0) .charAt(0);

        isKid = age < 12;
        isSenior = age >= 65;
        hasCoupon = reply == 'Y' || reply == 'y';
        hasNoCoupon = reply == 'N' || reply == 'n';

        if (!isKid && !isSenior) {
            price = 9.25;
        }
        if (isKid || isSenior) {
            price = 5.25;
        }

        if (hasCoupon) {
            price -= 2.00;
        }
        if (!hasCoupon && !hasNoCoupon) {
            System.out.println("Huh?");
        }

        System.out.print("Please pay $");
        System.out.print(price);
        System.out.print(". ");
        System.out.println("Enjoy the show!");
    }
}
```

Runs of the Listing 10-3 code look like the stuff in Figure 10-5. The only difference between Listings 10-2 and 10-3 is the use of `boolean` variables. In Listing 10-3, you get past all the less-than signs and double equal signs before the start of any `if` statements. By the time you encounter the two `if` statements, the conditions can use simple words — words like `isKid`, `isSenior`, and `hasCoupon`. You can read more about `boolean` variables in Chapter 8.

Adding a `boolean` variable can make your code more manageable. But some programming languages don't have `boolean` variables, so many programmers prefer to create `if` conditions on the fly. That's why I mix the two techniques (conditions with and without `boolean` variables) in this book.

Mixing different logical operators together

If you read about Listing 10-2, you know that my local movie theater offers discount coupons. The trouble is, I can't use a coupon along with any other discount. I tried to convince the ticket taker that I'm under 12 years of age, but he didn't buy it. When that didn't work, I tried combining the coupon with the senior citizen discount. That didn't work, either. The theater must use some software that checks for people like me. It looks something like the code in Listing 10-4. Take a look at Figure 10-6.

Listing 10-4: No Extra Break for Kids or Seniors

```
import java.util.Scanner;

class CheckAgeForDiscount {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        int age;
        double price = 0.00;
        char reply;

        System.out.print("How old are you? ");
        age = myScanner.nextInt();

        System.out.print("Have a coupon? (Y/N) ");
        reply = myScanner.findWithinHorizon(".", 0)
                                              .charAt(0);

        if (age >= 12 && age < 65) {
            price = 9.25;
        }
        if (age < 12 || age >= 65) {
            price = 5.25;
        }

        if ((reply == 'Y' || reply == 'y') &&
            (age >= 12 && age < 65)) {
            price -= 2.00;
        }


        System.out.print("Please pay $");
        System.out.print(price);
        System.out.print(". ");
        System.out.println("Enjoy the show!");
    }
}
```

```
How old are you? 7
Have a coupon? (Y/N) Y
Please pay $5.25. Enjoy the show!

How old are you? 25
Have a coupon? (Y/N) y
Please pay $7.25. Enjoy the show!

How old are you? 25
Have a coupon? (Y/N) n
Please pay $9.25. Enjoy the show!

How old are you? 85
Have a coupon? (Y/N) y
Please pay $5.25. Enjoy the show!

How old are you? 85
Have a coupon? (Y/N) Y
Please pay $5.25. Enjoy the show!
```

Figure 10-6: Running the code in Listing 10-4.

Listing 10-4 is a lot like its predecessors, Listings 10-1 and 10-2. The big difference is the bolded `if` statement. This `if` statement tests two things, and each thing has two parts of its own:

1. **Does the customer have a coupon?**

   That is, did the customer reply with either `Y` *or* with `y`?

2. **Is the customer in the regular age group?**

   That is, is the customer at least 12 years old *and* younger than 65?

In Listing 10-4, I join items 1 and 2 using the `&&` operator. I do this because both items (item 1 *and* item 2) must be true in order for the customer to qualify for the $2.00 discount, as illustrated in Figure 10-7.

Using parentheses

Listing 10-4 demonstrates something important about conditions. Sometimes, you need parentheses to make a condition work correctly. Take, for example, the following incorrect `if` statement:

```
//This code is incorrect:
if (reply == 'Y' || reply == 'y' &&
    age >= 12 && age < 65) {
    price -= 2.00;
}
```

Compare this code with the correct code in Listing 10-4. This incorrect code has no parentheses to group `reply == 'Y'` with `reply == 'y'`, or to group `age >= 12` with `age < 65`. The result is the bizarre pair of runs in Figure 10-8.

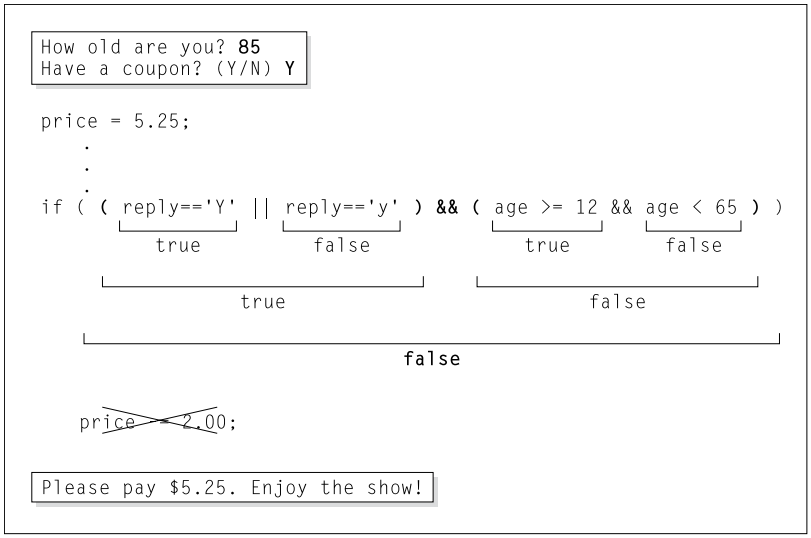


Figure 10-7: Both the reply and the age criteria must be true.

```
How old are you? 85
Have a coupon? (Y/N) y
Please pay $5.25. Enjoy the show!

How old are you? 85
Have a coupon? (Y/N) Y
Please pay $3.25. Enjoy the show!
```

Figure 10-8: A capital offense.

In Figure 10-8, notice that the `y` and `Y` inputs yield different ticket prices, even though the age is 85 in both runs. This happens because, without parentheses, any `&&` operator gets evaluated before any `||` operator. (That's the rule in the Java programming language — evaluate `&&` before `||`.) When `reply` is `Y`, the condition in the bad `if` statement takes the following form:

```
reply == 'Y' || some-other-stuff-that-doesn't-matter
```

Whenever `reply == 'Y'` is `true`, the whole condition is automatically `true`, as illustrated in Figure 10-9.

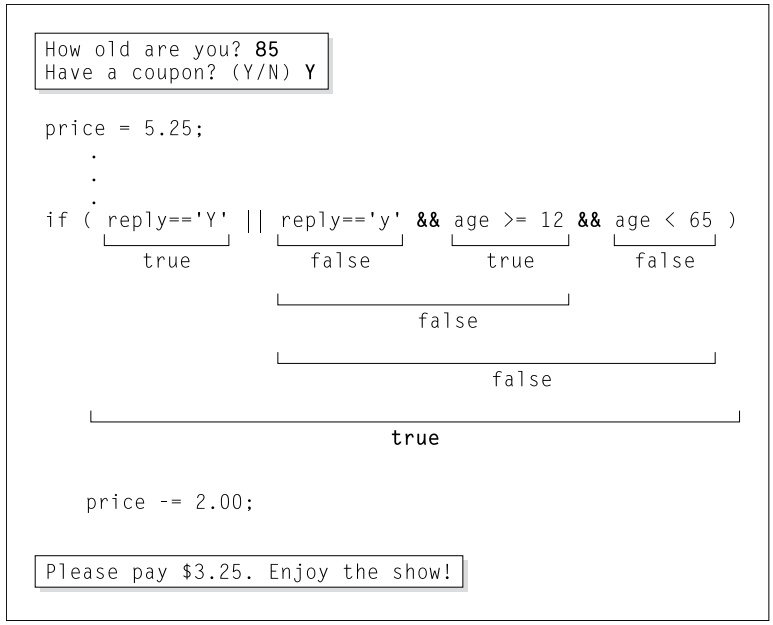


Figure 10-9: "True or false" makes "true."

Building a Nest

The year is 1968, and *The Prisoner* is on TV. In the last episode, the show's hero meets his nemesis, "Number One." At first, Number One wears a spooky happy-face/sad-face mask, and when the mask comes off, there's a monkey mask underneath. To find out what's behind the monkey mask, you have to watch the series on DVD. But in the meantime, notice the layering: a mask within a mask. You can do the same kind of thing with `if` statements. This section's example shows you how.

But first, take a look at Listing 10-4. In that code, the condition `age >= 12 && age < 65` is tested twice. Both times, the computer sends the numbers 12, 65, and the `age` value through its jumble of circuits, and both times, the computer gets the same answer. This is wasteful, but waste isn't your only concern.

What if you decide to change the age limit for senior tickets? From now on, no one under 100 gets a senior discount. You fish through the code and see the first `age >= 12 && age < 65` test. You change `65` to `100`, pat yourself on the back, and go home. The problem is, you've changed one of the two `age >= 12 && age < 65` tests, but you haven't changed the other. Wouldn't it be better to keep all the `age >= 12 && age < 65` testing in just one place?

Listing 10-5 comes to the rescue. In Listing 10-5, I smoosh all my `if` statements together into one big glob. The code is dense, but it gets the job done nicely.

Listing 10-5: Nested if Statements

```
import java.util.Scanner;

class AnotherAgeCheck {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        int age;
        double price = 0.00;
        char reply;

        System.out.print("How old are you? ");
        age = myScanner.nextInt();

        System.out.print("Have a coupon? (Y/N) ");
        reply = myScanner.findWithinHorizon(".", 0).charAt(0);

        if (age >= 12 && age < 65) {
            price = 9.25;
            if (reply == 'Y' || reply == 'y') {
                price -= 2.00;
            }
        } else {
            price = 5.25;
        }

        System.out.print("Please pay $");
        System.out.print(price);
        System.out.print(". ");
        System.out.println("Enjoy the show!");
    }
}
```

Nested if statements

A run of the code in Listing 10-5 looks identical to a run for Listing 10-4. You can see several runs in Figure 10-6. The main idea in Listing 10-5 is to put an `if` statement inside another `if` statement. After all, Chapter 9 says that an `if` statement can take the following form:

```
if (Condition) {
    SomeStatements
} else {
    OtherStatements
}
```

Who says *SomeStatements* can't contain an `if` statement? For that matter, *OtherStatements* can also contain an `if` statement. And, yes, you can create an `if` statement within an `if` statement within an `if` statement. There's no predefined limit on the number of `if` statements that you can have.

```
if (age >= 12 && age < 65) {
    price = 9.25;
    if (reply == 'Y' || reply == 'y') {
        if (isSpecialFeature) {
            price -= 1.00;
        } else {
            price -= 2.00;
        }
    }
} else {
    price = 5.25;
}
```

When you put one `if` statement inside another, you create *nested* `if` statements. Nested statements aren't difficult to write, as long as you take things slowly and keep a clear picture of the code's flow in your mind. If it helps, draw yourself a diagram like the one shown in Figure 10-10.

When you nest statements, you must be compulsive about the use of indentation and braces (see Figure 10-11). When code has misleading indentation, no one (not even the programmer who wrote the code) can figure out how the code works. A nested statement with sloppy indentation is a programmer's nightmare.

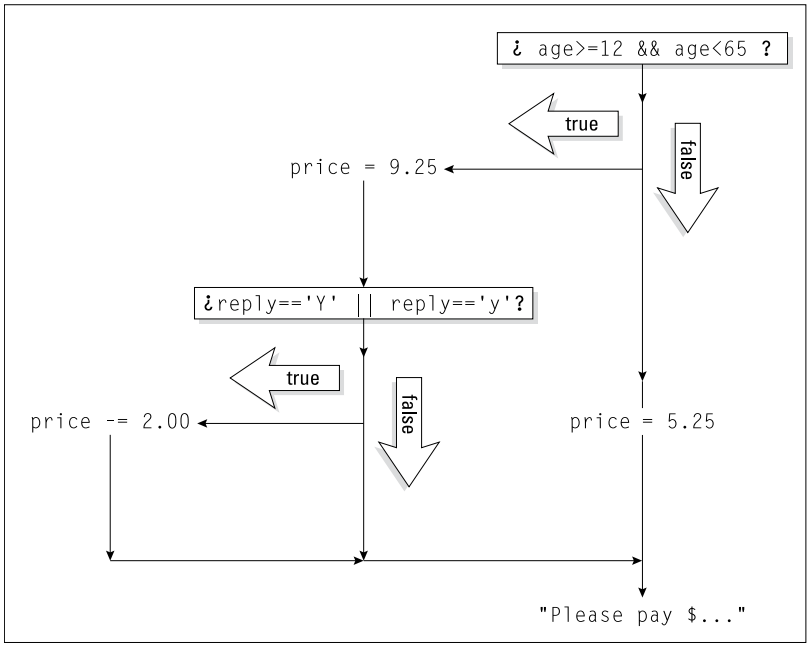


Figure 10-10: The flow in Listing 10-5.

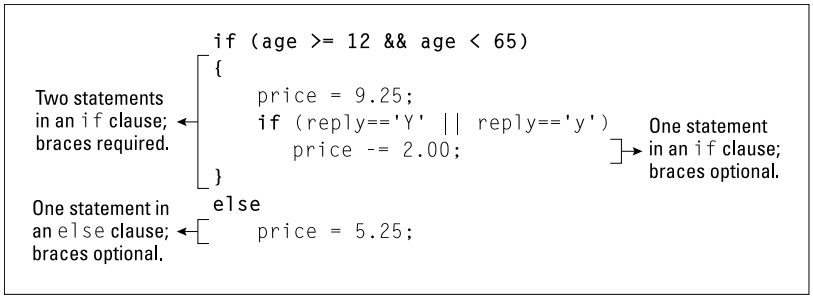

if (age >= 12 && age < 65)
{
price = 9.25;
if (reply=='Y' || reply=='y')
price -= 2.00;
}
else
price = 5.25;
Two statements in an if clause; braces required.
One statement in an if clause; braces optional.
One statement in an else clause; braces optional.

Figure 10-11: Be careful about adding the proper indentation and braces.

Cascading if statements

Here's a riddle: You have two baseball teams — the Hankees and the Socks. You want to display the teams' scores on two separate lines, with the winner's score coming first. (On the computer screen, the winner's score is displayed above the loser's score.) What happens when the scores are tied?

Do you give up? The answer is, there's no right answer. What happens depends on the way you write the program. Take a look back at Listing 9-4 in Chapter 9. When the scores are equal, the condition `hankees > socks` is `false`. So the program's flow of execution drops down to the `else` clause. That clause displays the Socks score first and the Hankees score second. (Refer to Figure 9-7.)

The program doesn't have to work this way. If I take Listing 9-4 and change `hankees > socks` to `hankees >= socks` then, in case of a tie, the Hankees score comes first.

Suppose that you want a bit more control. When the scores are equal, you want an `It's a tie` message. To do this, think in terms of a three-pronged fork. You have a prong for a Hankees win, another prong for a Socks win, and a third prong for a tie. You can write this code in several different ways, but one way that makes lots of sense is in Listing 10-6. For three runs of the code in Listing 10-6, see Figure 10-12.

Listing 10-6: In Case of a Tie . . .

```
import java.util.Scanner;
import static java.lang.System.out;

class WinLoseOrTie {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        int hankees, socks;

        out.print("Hankees and Socks scores?  ");
        hankees = myScanner.nextInt();
        socks = myScanner.nextInt();
        out.println();
```

```
        if (hankees > socks) {
            out.println("Hankees win...");
            out.print("Hankees: ");
            out.println(hankees);
            out.print("Socks:   ");
            out.println(socks);
        } else if (socks > hankees) {
            out.println("Socks win...");
            out.print("Socks:   ");
            out.println(socks);
            out.print("Hankees: ");
            out.println(hankees);
        } else {
            out.println("It's a tie...");
            out.print("Hankees: ");
            out.println(hankees);
            out.print("Socks:   ");
            out.println(socks);
        }
    }
}
```

```
Hankees and Socks scores?  9 4

Hankees win...
Hankees: 9
Socks:   4


Hankees and Socks scores?  3 8

Socks win...
Socks:   8
Hankees: 3


Hankees and Socks scores?  0 0

It's a tie...
Hankees: 0
Socks:   0
```

Figure 10-12: Go, team, go!

Listing 10-6 illustrates a way of thinking about a problem. You have one question with more than two answers. (In this section's baseball problem, the question is "Who wins?" and the answers are "Hankees," "Socks," or

"Neither.") The problem begs for an `if` statement, but an `if` statement has only two branches — the `true` branch and the `false` branch. So you combine alternatives to form *cascading if statements*.

In Listing 10-6, the format for the cascading `if` statements is

```
if (Condition1) {
    SomeStatements
} else if (Condition2) {
    OtherStatements
} else {
    EvenMoreStatements
}
```

In general, you can use `else if` as many times as you want:

```
if (hankeesWin) {
    out.println("Hankees win...");
    out.print("Hankees: ");
    out.println(hankees);
    out.print("Socks:   ");
    out.println(socks);
} else if (socksWin) {
    out.println("Socks win...");
    out.print("Socks:   ");
    out.println(socks);
    out.print("Hankees: ");
    out.println(hankees);
} else if (isATie) {
    out.println("It's a tie...");
    out.print("Hankees: ");
    out.println(hankees);
    out.print("Socks:   ");
    out.println(socks);
} else if (gameCancelled) {
    out.println("Sorry, sports fans.");
} else {
    out.println("The game isn't over yet.");
}
```

Nothing is special about cascading `if` statements. This isn't a new programming language feature. Cascading `if` statements take advantage of a loophole in Java — a loophole about omitting curly braces in certain circumstances. Other than that, cascading `if` statements just gives you a new way to think about decisions within your code.

Note: Listing 10-6 uses a static import declaration to avoid needless repetition of the words `System.out`. To read a little bit about the static import declaration (along with an apology for my not explaining this concept more thoroughly), see Chapter 9. Then to get the real story on static import declarations, see Chapter 18.

Enumerating the Possibilities

Chapter 8 describes Java's `boolean` type — the type with only two values (`true` and `false`). The `boolean` type is very handy, but sometimes you need more values. After all, a traffic light's values can be `green`, `yellow`, or `red`. A playing card's suit can be `spade`, `club`, `heart`, or `diamond`. And a weekday can be `Monday`, `Tuesday`, `Wednesday`, `Thursday`, or `Friday`.

Life is filled with small sets of possibilities, and Java has a feature that can reflect these possibilities. The feature is called an `enum` type. It's available from Java version 5.0 onward.

Creating an enum type

The story in Listing 10-6 has three possible endings — the Hankees win, the Socks win, or the game is tied. You can represent the possibilities with the following line of Java code:

```
enum WhoWins {home, visitor, neither}
```

This week's game is played at Hankeeville's SnitSoft Stadium, so the value `home` represents a win for the Hankees, and the value `visitor` represents a win for the Socks.

One of the goals in computer programming is for each program's structure to mirror whatever problem the program solves. When a program reminds you of its underlying problem, the program is easy to understand and inexpensive to maintain. For example, a program to tabulate customer accounts should use names like `customer` and `account`. And a program that deals with three possible outcomes (home wins, visitor wins, and tie) should have a variable with three possible values. The line `enum WhoWins {home, visitor, neither}` creates a type to store three values.

The `WhoWins` type is called an *enum type*. Think of the new `WhoWins` type as a `boolean` on steroids. Instead of two values (`true` and `false`), the `WhoWins` type has three values (`home`, `visitor`, and `neither`). You can create a variable of type `WhoWins`

```
WhoWins who;
```

and then assign a value to the new variable.

```
who = WhoWins.home;
```

In the next section, I put the `WhoWins` type to good use.

Using an enum type

Listing 10-7 shows you how to use the brand new `WhoWins` type.

Listing 10-7: Proud Winners and Sore Losers

```
import java.util.Scanner;
import static java.lang.System.out;

class Scoreboard {

    enum WhoWins {home, visitor, neither}

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        int hankees, socks;
        WhoWins who;

        out.print("Hankees and Socks scores?  ");
        hankees = myScanner.nextInt();
        socks = myScanner.nextInt();
        out.println();

        if (hankees > socks) {
            who = WhoWins.home;
            out.println("The Hankees win :-)");
        } else if (socks > hankees) {
            who = WhoWins.visitor;
            out.println("The Socks win :-(");
        } else {
            who = WhoWins.neither;
            out.println("It's a tie :-|");
        }

        out.println();
        out.println("Today's game is brought to you by");
        out.println("SnitSoft, the number one software");
        out.println("vendor in the Hankeeville area.");
        out.println("SnitSoft is featured proudly in");
        out.println("Chapter 6. And remember, four out");
        out.println("of five doctors recommend");
        out.println("SnitSoft to their patients.");
        out.println();

        if (who == WhoWins.home) {
            out.println("We beat 'em good. Didn't we?");
        }

        if (who == WhoWins.visitor) {
            out.println("The umpire made an unfair");
            out.println("call.");
        }
        if (who == WhoWins.neither) {
            out.println("The game goes into overtime.");
        }
    }
}
```

Three runs of the program in Listing 10-7 are pictured in Figure 10-13. Here's what happens in Listing 10-7:

- **I create a variable to store values of type `WhoWins`.**

  Just as the line

  ```
  double amount;
  ```

  declares `amount` to store `double` values (values like 5.95 and 30.95), the line

  ```
  WhoWins who;
  ```

  declares `who` to store `WhoWins` values (values like `home`, `visitor`, and `neither`).

- **I assign a value to the `who` variable.**

  I execute one of the

  ```
  who = WhoWins.something;
  ```

  assignment statements. The statement that I execute depends on the outcome of the `if` statement's `hankees > socks` comparison.

  Notice that I refer to each of the `WhoWins` values in Listing 10-7. I write `WhoWins.home`, `WhoWins.visitor`, or `WhoWins.neither`. If I forget the `WhoWins` prefix and type

  ```
  who = home;  //This assignment doesn't work!
  ```

  then the compiler gives me a `home cannot be resolved to a variable` error message. That's just the way `enum` types work.

- **I compare the variable's value with each of the `WhoWins` values.**

  In one `if` statement, I check the `who == WhoWins.home` condition. In the remaining two `if` statements, I check for the other `WhoWins` values.

Near the end of Listing 10-7, I could have done without `enum` values. I could have tested things like `hankees > socks` a second time.

```
if (hankees > socks) {
    out.println("The Hankees win :-)");
}

// And later in the program...

if (hankees > socks) {
    out.println("We beat 'em good. Didn't we?");
}
```

But that tactic would be clumsy. In a more complicated program, I may end up checking `hankees > socks` a dozen times. It would be like asking the same question over and over again.

Instead of repeatedly checking the `hankees > socks` condition, I store the game's outcome as an enum value. Then I check the enum value as many times as I want. That's a very tidy way to solve the repeated checking problem.

```
Hankees and Socks scores?  9 4

The Hankees win :-)

Today's game is brought to you by
SnitSoft, the number one software
vendor in the Hankeeville area.
SnitSoft is featured proudly in
Chapter 6. And remember, four out
of five doctors recommend
SnitSoft to their patients.

We beat 'em good. Didn't we?


Hankees and Socks scores?  3 8

The Socks win :-(

Today's game is brought to you by
SnitSoft, the number one software
vendor in the Hankeeville area.
SnitSoft is featured proudly in
Chapter 6. And remember, four out
of five doctors recommend
SnitSoft to their patients.

The umpire made an unfair call.


Hankees and Socks scores?  0 0

It's a tie :-|

Today's game is brought to you by
SnitSoft, the number one software
vendor in the Hankeeville area.
SnitSoft is featured proudly in
Chapter 6. And remember, four out
of five doctors recommend
SnitSoft to their patients.

The game goes into overtime.
```

Figure 10-13: Joy in Hankeeville?

Chapter 11

How to Flick a Virtual Switch

In This Chapter

- Dealing with many alternatives
- Jumping out from the middle of a statement
- Handling alternative assignments

Imagine playing *Let's Make a Deal* with ten different doors. "Choose door number 1, door number 2, door number 3, door number 4. . . . Wait! Let's break for a commercial. When we come back, I'll say the names of the other six doors."

What Monty Hall needs is Java's `switch` statement.

Meet the Switch Statement

The code in Listing 9-2 in Chapter 9 simulates a fortune-telling toy — an electronic oracle. Ask the program a question, and the program randomly generates a yes or no answer. But, as toys go, the code in Listing 9-2 isn't much fun. The code has only two possible answers. There's no variety. Even the earliest talking dolls could say about ten different sentences.

Suppose that you want to enhance the code of Listing 9-2. The call to `myRandom.nextInt(10) + 1` generates numbers from 1 to 10. So maybe you can display a different sentence for each of the ten numbers. A big pile of `if` statements should do the trick:

```
if (randomNumber == 1) {
    System.out.println("Yes. Isn't it obvious?");
}
if (randomNumber == 2) {
    System.out.println("No, and don't ask again.");
}
```

```
if (randomNumber == 3) {
    System.out.print("Yessir, yessir!");
    System.out.println(" Three bags full.");
}
if (randomNumber == 4)
    .
    .
    .
if (randomNumber < 1 || randomNumber > 10) {
    System.out.print("Sorry, the electronic oracle");
    System.out.println(" is closed for repairs.");
}
```

But that approach seems wasteful. Why not create a statement that checks the value of `randomNumber` just once and then takes an action based on the value that it finds? Fortunately, just such a statement exists: the *switch* statement. Listing 11-1 has an example of a `switch` statement.

Listing 11-1: An Answer for Every Occasion

```
import java.util.Scanner;
import java.util.Random;
import static java.lang.System.out;

class TheOldSwitcheroo {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        Random myRandom = new Random();
        int randomNumber;

        out.print("Type your question, my child:  ");
        myScanner.nextLine();

        randomNumber = myRandom.nextInt(10) + 1;

        switch (randomNumber) {
        case 1:
            out.println("Yes. Isn't it obvious?");
            break;

        case 2:
            out.println("No, and don't ask again.");
            break;
        case 3:
```

```
            out.print("Yessir, yessir!");
             out.println(" Three bags full.");
            break;

        case 4:
            out.print("What part of 'no'");
            out.println(" don't you understand?");
            break;

        case 5:
            out.println("No chance, Lance.");
            break;

        case 6:
            out.println("Sure, whatever.");
            break;

        case 7:
            out.print("Yes, but only if");
            out.println(" you're nice to me.");
            break;

        case 8:
            out.println("Yes (as if I care).");
            break;

        case 9:
            out.print("No, not until");
            out.println(" Cromwell seizes Dover.");
            break;

        case 10:
            out.print("No, not until");
            out.println(" Nell squeezes Rover.");
            break;

        default:
            out.print("You think you have");
            out.print(" problems?");
            out.print(" My random number");
            out.println(" generator is broken!");
            break;
        }

        out.println("Goodbye");
    }
}
```

The cases in a switch statement

Figure 11-1 shows three runs of the program in Listing 11-1. Here's what happens during one of these runs:

- The user types a heavy question, and the variable `randomNumber` gets a value. In the second run of Figure 11-1, this value is 2.
- Execution of the code in Listing 11-1 reaches the top of the `switch` statement, so the computer starts checking this statement's `case` clauses. The value 2 doesn't match the topmost `case` clause (the `case 1` clause), so the computer moves on to the next `case` clause.
- The value in the next `case` clause (the number 2) matches the value of the `randomNumber` variable, so the computer executes the statements in this `case 2` clause. These two statements are

  ```
  out.println("No, and don't ask again.");
  break;
  ```

  The first of the two statements displays `No, and don't ask again` on the screen. The second statement is called a *break* statement. (What a surprise!) When the computer encounters a `break` statement, the computer jumps out of whatever `switch` statement it's in. So in Listing 11-1, the computer skips right past `case 3`, `case 4`, and so on. The computer jumps to the statement just after the end of the `switch` statement.
- The computer displays `Goodbye` because that's what the statement after the `switch` statement tells the computer to do.

```
Type your question, my child:  Is the Continuum Hypothesis true?
Sure, whatever.
Goodbye

Type your question, my child:  Does P=NP?
No, and don't ask again.
Goodbye

Type your question, my child:  Does Turing machine T halt on input i?
Yes, but only if you're nice to me.
Goodbye
```

Figure 11-1: Running the code of Listing 11-1.

The overall idea behind the program in Listing 11-1 is illustrated in Figure 11-2.

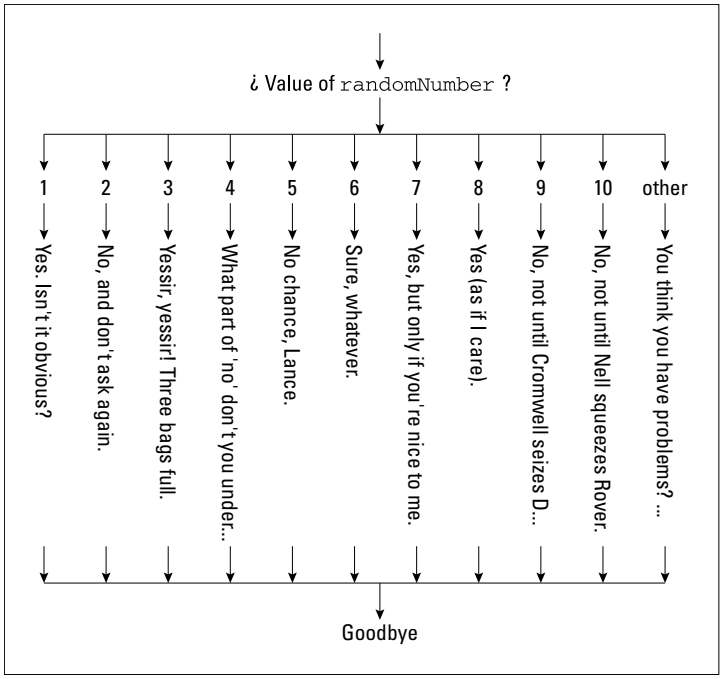


Figure 11-2: A fork with 11 prongs.

The default in a switch statement

What if something goes terribly wrong during a run of the Listing 11-1 program? Suppose that the expression `myRandom.nextInt(10) + 1` generates a number that's not in the 1 to 10 range. Then the computer responds by dropping past all the `case` clauses. Instead of landing on a `case` clause, the computer jumps to the `default` clause. In the `default` clause, the computer displays `You think you have problems?...`, and then breaks out of the `switch` statement. After the computer is out of the `switch` statement, the computer displays `Goodbye`.

You don't really need to put a `break` at the very end of a `switch` statement. In Listing 11-1, the last `break` (the `break` that's part of the `default` clause) is just for the sake of overall tidiness.

Picky details about the switch statement

A `switch` statement can take the following form:

```
switch (Expression) {
case FirstValue:
    Statements

case SecondValue:
    MoreStatements

// ... more cases...


default:
    EvenMoreStatements
}
```

Here are some tidbits about `switch` statements:

- The *Expression* doesn't have to have an `int` value. It can be `char`, `byte`, `short`, or `int`.

  For example, the following code works in Java 5 or later:

  ```
  char letterGrade;
  letterGrade =
      myScanner.findWithinHorizon(".",0).charAt(0);

  switch (letterGrade) {
  case 'A':
      System.out.println("Excellent");
      break;

  case 'B':
      System.out.println("Good");
      break;

  case 'C':
      System.out.println("Average");
      break;
  }
  ```

  In fact, if you avoid using the `Scanner` class and its `findWithin Horizon` method, then this bullet's `switch` statement works with all versions of Java — old and new.

- If you use Java 7 or later, you the *Expression* can be a `String`. For example, the following code doesn't work with Java 6, but works well in Java 7:

  ```
  String description;
  description = myScanner.next();

  switch (description) {
  case "Excellent":
      System.out.println('A');
      break;
  ```

```
case "Good":
    System.out.println('B');
    break;

case "Average":
    System.out.println('C');
    break;
}
```

- The *Expression* doesn't have to be a single variable. It can be any expression of type `char`, `byte`, `short`, or `int`. For example, you can simulate the rolling of two dice with the following code:

```
int die1, die2;

die1 = myRandom.nextInt(6) + 1;
die2 = myRandom.nextInt(6) + 1;

switch (die1 + die2) {
   //...etc.
```

- The cases in a `switch` statement don't have to be in order. Here's some acceptable code:

```
switch (randomNumber) {
case 2:
    System.out.println("No, and don't ask again.");
    break;

case 1:
    System.out.println("Yes. Isn't it obvious?");
    break;

case 3:
    System.out.print("Yessir, yessir!");
    System.out.println(" Three bags full.");
    break;

//...etc.
```

This mixing of cases may slow you down when you're trying to read a program, but it's legal nonetheless.

- You don't need a `case` for each expected value of the *Expression*. You can leave some expected values to the `default`. Here's an example:

```
switch (randomNumber) {
case 1:
    System.out.println("Yes. Isn't it obvious?");
    break;

case 5:
    System.out.println("No chance, Lance.");
    break;
```

```
case 7:
    System.out.print("Yes, but only if");
    System.out.println(" you're nice to me.");
    break;

case 10:
    System.out.print("No, not until");
    System.out.println(" Nell squeezes Rover.");
    break;

default:
    System.out.print("Sorry,");
    System.out.println(" I just can't decide.");
    break;
}
```

- The `default` clause is optional.

```
switch (randomNumber) {
case 1:
    System.out.println("Yes. Isn't it obvious?");
    break;

case 2:
    System.out.println("No, and don't ask again.");
    break;

case 3:
    System.out.print("I'm too tired.");
    System.out.println(" Go ask somebody else.");
}
System.out.println("Goodbye");
```

If you have no `default` clause, and a value that's not covered by any of the cases comes up, then the `switch` statement does nothing. For example, if `randomNumber` is 4, then the preceding code displays `Goodbye`, and nothing else.

- In some ways, `if` statements are more versatile than `switch` statements. For example, you can't use a condition in a `switch` statement's *`Expression`*:

```
//You can't do this:
switch (age >= 12 && age < 65)
```

You can't use a condition as a `case` value, either:

```
//You can't do this:
switch (age) {
case age <= 12: //...etc.
```

To break or not to break

At one time or another, every Java programmer forgets to use `break` statements. At first, the resulting output is confusing, but then the programmer remembers fall-through. The term *fall-through* describes what happens when you end a `case` without a `break` statement. What happens is that execution of the code falls right through to the next `case` in line. Execution keeps falling through until you eventually reach a `break` statement or the end of the entire `switch` statement.

If you don't believe me, just look at Listing 11-2. This listing's code has a `switch` statement gone bad.

Listing 11-2: Please, Gimme a Break!

```
/*
 * This isn't good code. The programmer forgot some
 * of the break statements.
 */
import java.util.Scanner;
import java.util.Random;
import static java.lang.System.out;

class BadBreaks {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        Random myRandom = new Random();
        int randomNumber;

        out.print("Type your question, my child:  ");
        myScanner.nextLine();

        randomNumber = myRandom.nextInt(10) + 1;

        switch (randomNumber) {
        case 1:
            out.println("Yes. Isn't it obvious?");

        case 2:
            out.println("No, and don't ask again.");

        case 3:
            out.print("Yessir, yessir!");
            out.println(" Three bags full.");
```

(continued)

Listing 11-2 ***(continued)***

```
            case 4:
                out.print("What part of 'no'");
                out.println(" don't you understand?");
                break;

            case 5:
                out.println("No chance, Lance.");

            case 6:
                out.println("Sure, whatever.");

            case 7:
                out.print("Yes, but only if");
                out.println(" you're nice to me.");

            case 8:
                out.println("Yes (as if I care).");

            case 9:
                out.print("No, not until");
                out.println(" Cromwell seizes Dover.");

            case 10:
                out.print("No, not until");
                out.println(" Nell squeezes Rover.");

            default:
                out.print("You think you have");
                out.print(" problems?");
                out.print(" My random number");
                out.println(" generator is broken!");
            }

            out.println("Goodbye");
        }
    }
```

I've put two runs of this code in Figure 11-3. In the first run, the `randomNumber` is 7. The program executes cases 7 through 10, and the `default`. In the second run, the `randomNumber` is 3. The program executes cases 3 and 4. Then, because case 4 has a `break` statement, the program jumps out of the `switch` and displays `Goodbye`.

The `switch` statement in Listing 11-2 is missing some `break` statements. Even without these `break` statements, the code compiles with no errors. But when you run the code in Listing 11-2, you don't get the results that you want.

```
Type your question, my child:  Do good things happen to good people?
Yes, but only if you're nice to me.
Yes (as if I care).
No, not until Cromwell seizes Dover.
No, not until Nell squeezes Rover.
You think you have problems? My random number generator is broken!
Goodbye

Type your question, my child:  Is your switch statement missing some breaks?
Yessir, yessir! Three bags full.
What part of 'no' don't you understand?
Goodbye
```

Figure 11-3: Please make up your mind.

Using Fall-Through to Your Advantage

Often, when you're using a `switch` statement, you don't want fall-through, so you pepper `break` statements throughout the `switch`. But, sometimes, fall-through is just the thing you need.

Take the number of days in a month. Is there a simple rule for this? Months containing the letter "r" have 31 days? Months in which "i" comes before "e" except after "c" have 30 days?

You can fiddle with `if` conditions all you want. But to handle all the possibilities, I prefer a `switch` statement. Listing 11-3 demonstrates the idea.

Listing 11-3: Finding the Number of Days in a Month

```
import java.util.Scanner;

class DaysInEachMonth {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        int month, numberOfDays = 0;
        boolean isLeapYear;

        System.out.print("Which month? ");
        month = myScanner.nextInt();

        switch (month) {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
```

(continued)

Listing 11-3 ***(continued)***

```
            case 10:
            case 12:
                numberOfDays = 31;
                break;

            case 4:
            case 6:
            case 9:
            case 11:
                numberOfDays = 30;
                break;

            case 2:
                System.out.print("Leap year (true/false)? ");
                isLeapYear = myScanner.nextBoolean();
                if (isLeapYear) {
                   numberOfDays = 29;
                } else {
                   numberOfDays = 28;
                }
            }

            System.out.print(numberOfDays);
            System.out.println(" days");
        }
    }
```

Figure 11-4 shows several runs of the program in Listing 11-3. For month number 6, the computer jumps to `case 6`. There are no statements inside the `case 6` clause, so that part of the program's run is pretty boring.

```
Which month? 1
31 days

Which month? 6
30 days

Which month? 2
Leap year (true/false)? false
28 days

Which month? 2
Leap year (true/false)? true
29 days
```

Figure 11-4: How many days until the next big deadline?

But with no `break` in the `case 6` clause, the computer marches right along to `case 9`. Once again, the computer finds no statements and no `break`, so the computer ventures to the next case, which is `case 11`. At that point, the computer hits pay dirt. The computer assigns 30 to `numberOfDays`, and breaks out of the entire `switch` statement (see Figure 11-5).

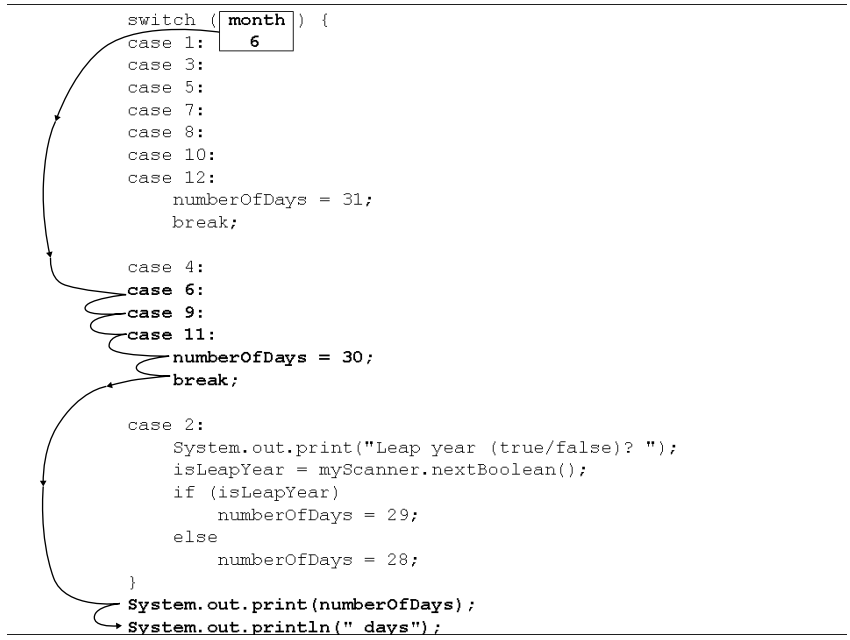


Figure 11-5: Follow the bouncing ball.

February is the best month of all. For one thing, the February case in Listing 11-3 contains a call to the `Scanner` class's `nextBoolean` method. The method expects me to type either `true` or `false`. The code uses whatever word I type to assign a value to a `boolean` variable. (In Listing 11-3, I assign `true` or `false` to the `isLeapYear` variable.)

February also contains its own `if` statement. In Chapter 10, I nest `if` statements within other `if` statements. But in February, I nest an `if` statement within a `switch` statement. That's cool.

Using a Conditional Operator

Java has a neat feature that I can't resist writing about. Using this feature, you can think about alternatives in a very natural way.

And what do I mean by "a natural way?" If I think out loud as I imitate the `if` statement near the end of Listing 11-3, I come up with this:

```
//The thinking in Listing 11-3:
What should I do next?
If this is a leap year,
   I'll make the numberOfDays be 29;
Otherwise,
   I'll make the numberOfDays be 28.
```

I'm wandering into an `if` statement without a clue about what I'm doing next. That seems silly. It's February, and everybody knows what you do in February. You ask how many days the month has.

In my opinion, the code in Listing 11-3 doesn't reflect the most natural way to think about February. So here's a more natural way:

```
//A more natural way to think about the problem:
The value of numberOfDays is...
   Wait! Is this a leap year?
      If yes, 29
      If no, 28
```

In this second, more natural way of thinking, I know from the start that I'm picking a number of days. So by the time I reach a fork in the road (`Is this a leap year?`), the only remaining task is to choose between 29 and 28.

I can make the choice with finesse:

```
case 2:
    System.out.print("Leap year (true/false)? ");
    isLeapYear = myScanner.nextBoolean();
    numberOfDays = isLeapYear ? 29 : 28;
```

The `? :` combination is called a *conditional operator*. In Figure 11-6, I show you how my natural thinking about February can morph into the conditional operator's format.

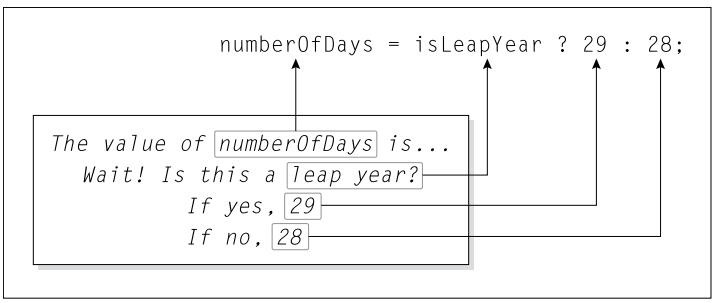


Figure 11-6: From your mind to the computer's code.

Taken as a whole, `isLeapYear ? 29 : 28` is an expression with a value. And what value does this expression have? Well, the value of `isLeapYear ? 29 : 28` is either 29 or 28. It depends on whether `isLeapYear` is or isn't true. That's how the conditional operator works:

- If the stuff before the question mark is `true`, then the whole expression's value is whatever comes between the question mark and the colon.
- If the stuff before the question mark is `false`, then the whole expression's value is whatever comes after the colon.

Figure 11-7 gives you goofy way to visualize these ideas.

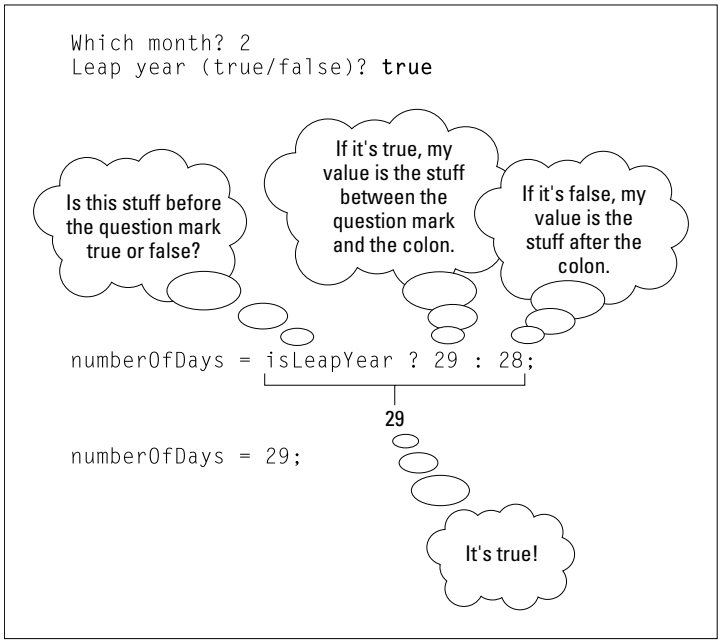

Which month? 2
Leap year (true/false)? true
Is this stuff before the question mark true or false?
If it's true, my value is the stuff between the question mark and the colon.
If it's false, my value is the stuff after the colon.
numberOfDays = isLeapYear ? 29 : 28;
29
numberOfDays = 29;
It's true!

Figure 11-7: Have you ever seen an expression talking to itself?

So the conditional operator's overall effect is as if the computer is executing

```
numberOfDays = 29;
```

or

```
numberOfDays = 28;
```

One way or another, `numberOfDays` gets a value, and the code solves the problem with style.

Chapter 12

Around and Around It Goes

In This Chapter

- Creating program loops
- Formulating solutions to problems with loops
- Diagnosing loop problems

Chapter 8 has code to reverse the letters in a four-letter word that the user enters. In case you haven't read Chapter 8 or you just don't want to flip to it, here's a quick recap of the code:

```
c1 = myScanner.findWithinHorizon(".",0).charAt(0);
c2 = myScanner.findWithinHorizon(".",0).charAt(0);
c3 = myScanner.findWithinHorizon(".",0).charAt(0);
c4 = myScanner.findWithinHorizon(".",0).charAt(0);

System.out.print(c4);
System.out.print(c3);
System.out.print(c2);
System.out.print(c1);
```

The code is just dandy for words with exactly four letters, but how do you reverse a five-letter word? As the code stands, you have to add two new statements:

```
c1 = myScanner.findWithinHorizon(".",0).charAt(0);
c2 = myScanner.findWithinHorizon(".",0).charAt(0);
c3 = myScanner.findWithinHorizon(".",0).charAt(0);
c4 = myScanner.findWithinHorizon(".",0).charAt(0);
c5 = myScanner.findWithinHorizon(".",0).charAt(0);

System.out.print(c5);
System.out.print(c4);
System.out.print(c3);
System.out.print(c2);
System.out.print(c1);
```

What a drag! You add statements to a program whenever the size of a word changes! You remove statements when the input shrinks! That can't be the best way to solve the problem. Maybe you can command a computer to add statements automatically. (But then again, maybe you can't.)

As luck would have it, you can do something that's even better. You can write a statement once and tell the computer to execute the statement many times. How many times? You can tell the computer to execute a statement as many times as it needs to be executed.

That's the big idea. The rest of this chapter has the details.

Repeating Instructions Over and Over Again (Java while Statements)

Here's a simple dice game: Keep rolling two dice until you roll 7 or 11. Listing 12-1 has a program that simulates the action in the game, and Figure 12-1 shows two runs of the program.

Listing 12-1: Roll 7 or 11

```
import java.util.Random;
import static java.lang.System.out;

class SimpleDiceGame {

    public static void main(String args[]) {
        Random myRandom = new Random();
        int die1 = 0, die2 = 0;

        while (die1 + die2 != 7 && die1 + die2 != 11) {
            die1 = myRandom.nextInt(6) + 1;
            die2 = myRandom.nextInt(6) + 1;

            out.print(die1);
            out.print(" ");
            out.println(die2);
        }

        out.print("Rolled ");
        out.println(die1 + die2);
    }
}
```

```
3 1
4 3
Rolled 7

2 1
4 6
5 3
6 4
4 6
1 5
2 2
1 5
1 3
2 6
1 4
6 5
Rolled 11
```

Figure 12-1: Momma needs a new pair of shoes.

At the core of Listing 12-1 is a thing called a *while statement* (also known as a *while loop*). A `while` statement has the following form:

```
while (Condition) {
   Statements
}
```

Rephrased in English, the `while` statement in Listing 12-1 would say

```
while the sum of the two dice isn't 7 and isn't 11
keep doing all the stuff in curly braces: {


}
```

The stuff in curly braces (the stuff that is repeated over and over again) is the code that gets two new random numbers and displays those random numbers' values. The statements in curly braces are repeated as long as `die1 + die2 != 7 && die1 + die2 != 11` keeps being true.

Each repetition of the statements in the loop is called an *iteration* of the loop. In Figure 12-1, the first run has 2 iterations, and the second run has 12 iterations.

When `die1 + die2 != 7 && die1 + die2 != 11` is no longer true (that is, when the `sum` is either 7 or 11), then the repeating of statements stops dead in its tracks. The computer marches on to the statements that come after the loop.

Following the action in a loop

To trace the action of the code in Listing 12-1, I'll borrow numbers from the first run in Figure 12-1:

- At the start, the values of `die1` and `die2` are both 0.
- The computer gets to the top of the `while` statement and checks to see if `die1 + die2 != 7 && die1 + die2 != 11` is true (see Figure 12-2). The condition is true, so the computer takes the `true` path in Figure 12-3.

  The computer performs an iteration of the loop. During this iteration, the computer gets new values for `die1` and `die2` and prints those values on the screen. In the first run of Figure 12-1, the new values are 3 and 1.

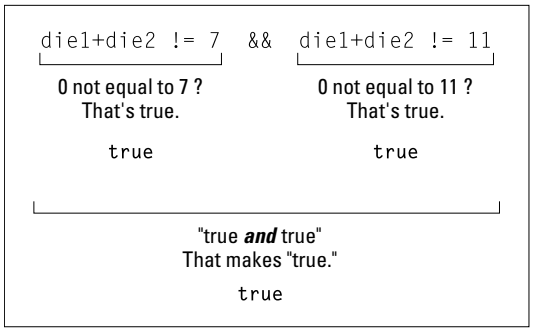


Figure 12-2: Two wrongs don't make a right, but two trues make a true.

- The computer returns to the top of the `while` statement and checks to see if `die1 + die2 != 7 && die1 + die2 != 11` is still true. The condition is true, so the computer takes the `true` path in Figure 12-3.

  The computer performs another iteration of the loop. During this iteration, the computer gets new values for `die1` and `die2` and prints those values on the screen. In Figure 12-1, the new values are 4 and 3.
- The computer returns to the top of the `while` statement and checks to see if `die1 + die2 != 7 && die1 + die2 != 11` is still true. Lo and behold! This condition has become false! (See Figure 12-4.) The computer takes the `false` path in Figure 12-3.

  The computer leaps to the statements after the loop. The computer displays `Rolled 7` and ends its run of the program.

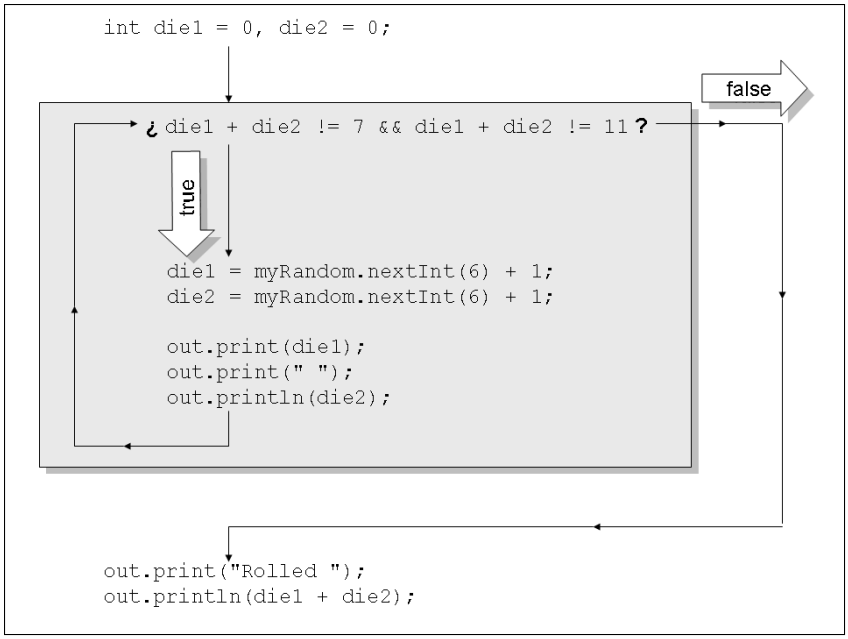


Figure 12-3: Paths through the code in Listing 12-1.

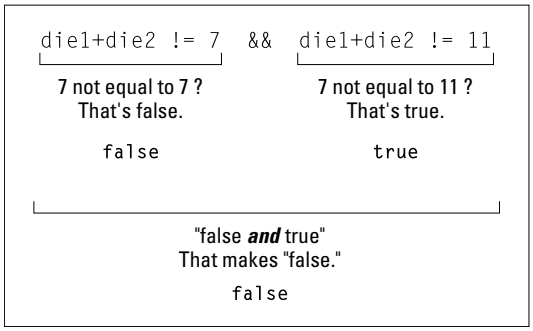

die1+die2 != 7 && die1+die2 != 11
7 not equal to 7 ? That's false.
false
7 not equal to 11 ? That's true.
true
"false and true" That makes "false."
false

Figure 12-4: Look! I rolled a seven!

No early bailout

In Listing 12-1, when the computer finds `die1 + die2 != 7 && die1 + die2 != 11` to be true, the computer marches on and executes all five statements inside the loop's curly braces. The computer executes

```
die1 = myRandom.nextInt(6) + 1;
die2 = myRandom.nextInt(6) + 1;
```

Statements and blocks (plagiarizing my own sentences from Chapter 9)

Java's `while` statements have a lot in common with `if` statements. Like an `if` statement, a `while` statement is a *compound statement*—that is, a `while` statement includes other statements within it. Also, both `if` statements and `while` statements typically include *blocks* of statements. When you surround a bunch of statements with curly braces, you get what's called a *block,* and a block behaves, in all respects, like a single statement.

In a typical `while` statement, you want the computer to repeat several smaller statements over and over again. To repeat several smaller statements, you combine those statements into one big statement. To do this, you make a block using curly braces.

In Listing 12-1, the block

```
{
  die1=myRandom.nextInt(6)+1;
  die2=myRandom.nextInt(6)+1;

  out.print(die1);
  out.print(" ");
  out.println(die2);
}
```

is a single statement. It's a statement that has, within it, five smaller statements. So this big block (this single statement) serves as one big statement inside the `while` statement in Listing 12-1.

That's the story about `while` statements and blocks. To find out how this stuff applies to `if` statements, see the "Statements and blocks" sidebar near the end of Chapter 9.

Maybe (just maybe), the new values of `die1` and `die2` add up to 7. Even so, the computer doesn't jump out in mid-loop. The computer finishes the iteration and executes

```
out.print(die1);
out.print(" ");
out.println(die2);
```

one more time. The computer performs the test again (to see if `die1 + die2 != 7 && die1 + die2 != 11` is still true) only after it fully executes all five statements in the loop.

Thinking about Loops (What Statements Go Where)

Here's a simplified version of the card game Twenty-One: You keep taking cards until the total is 21 or higher. Then, if the total is 21, you win. If the

total is higher, you lose. (By the way, each face card counts as a 10.) To play this game, you want a program whose runs look like the runs in Figure 12-5.

In most sections of this book, I put a program's listing before the description of the program's features. But this section is different. This section deals with strategies for composing code. So in this section, I start by brainstorming about strategies.

```
Card    Total
8       8
6       14
3       17
4       21
You win :-)

Card    Total
1       1
7       8
3       11
4       15
2       17
2       19
3       22
You lose :-(
```

Figure 12-5: You win sum; you lose sum.

Finding some pieces

How do you write a program that plays a simplified version of Twenty-One? I start by fishing for clues in the game's rules, spelled out in this section's first paragraph. The big fishing expedition is illustrated in Figure 12-6.

With the reasoning in Figure 12-6, I need a loop and an `if` statement:

```
while (total < 21) {
    //do stuff
}

if (total == 21) {
    //You win
} else {
    //You lose
}
```

What else do I need to make this program work? Look at the sample output in Figure 12-5. I need a heading with the words `Card` and `Total`. That's a call to `System.out.println`:

```
System.out.println("Card    Total");
```

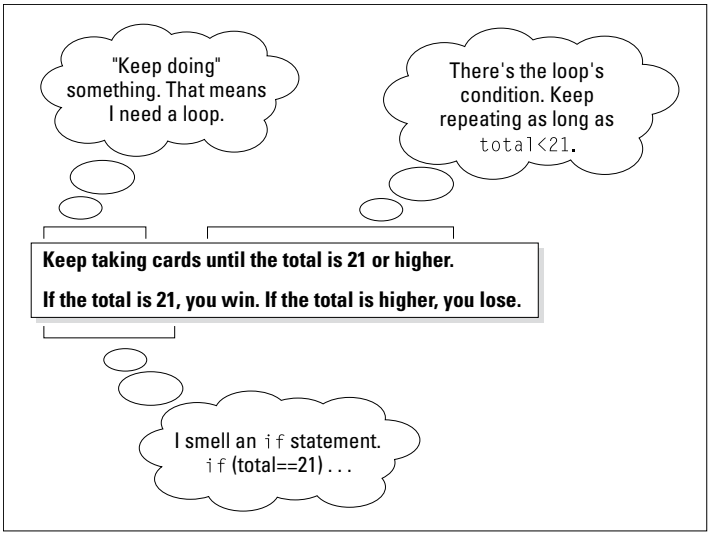

"Keep doing" something. That means I need a loop.
There's the loop's condition. Keep repeating as long as total<21.
Keep taking cards until the total is 21 or higher.
If the total is 21, you win. If the total is higher, you lose.
I smell an if statement. if (total==21) . . .

Figure 12-6: Thinking about a programming problem.

I also need several lines of output — each containing two numbers. For example, in Figure 12-5, the line `6 14` displays the values of two variables. One variable stores the most recently picked card; the other variable stores the total of all cards picked so far:

```
System.out.print(card);
System.out.print("        ");
System.out.println(total);
```

Now I have four chunks of code, but I haven't decided how they all fit together. Well, you can go right ahead and call me crazy. But at this point in the process, I imagine those four chunks of code circling around one another, like part of a dream sequence in a low-budget movie. As you may imagine, I'm not very good at illustrating circling code in dream sequences. Even so, I handed my idea to the art department folks at Wiley Publishing, and they came up with the picture in Figure 12-7.

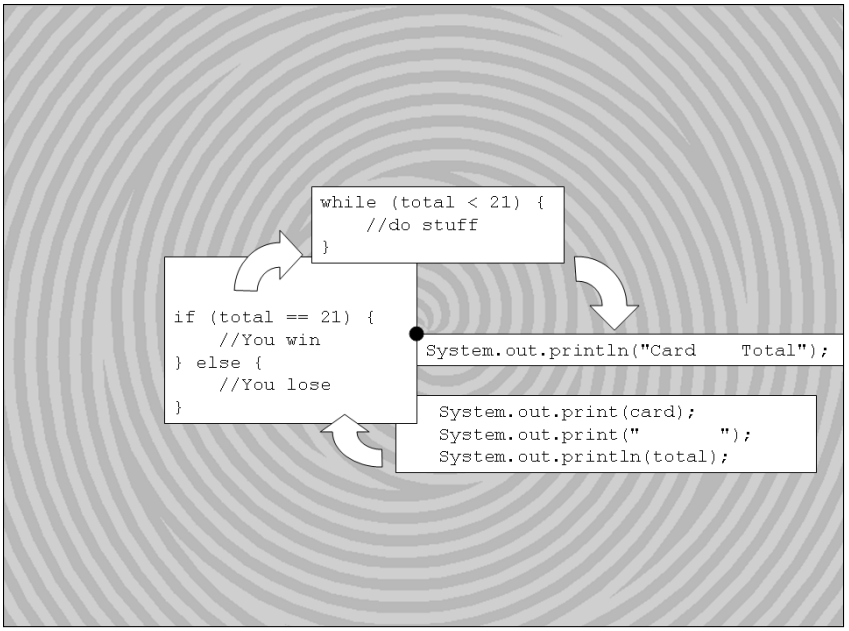


Figure 12-7: . . . and where they stop, nobody knows.

Assembling the pieces

Where should I put each piece of code? The best way to approach the problem is to ask how many times each piece of code should be executed:

- **The program displays `card` and `total` values more than once.** For example, in the first run of Figure 12-5, the program displays these values four times (first `8 8`, then `6 14`, and so on). To get this repeated display, I put the code that creates the display inside the loop:

```
while (total < 21) {
    System.out.print(card);
    System.out.print("       ");
    System.out.println(total);
}
```

- **The program displays the `Card Total` heading only once per run.** This display comes before any of the repeated number displays, so I put the heading code before the loop:

```
System.out.println("Card    Total");

while (total < 21) {
    System.out.print(card);
    System.out.print("       ");
    System.out.println(total);
}
```

- **The program displays `You win` or `You lose` only once per run.** This message display comes after the repeated number displays. So I put the win/lose code after the loop:

```
//Preliminary draft code - NOT ready for prime time:
System.out.println("Card    Total");

while (total < 21) {
    System.out.print(card);
    System.out.print("       ");
    System.out.println(total);
}

if (total == 21) {
    System.out.println("You win :-)");
} else {
    System.out.println("You lose :-(");
}
```

Getting values for variables

I almost have a working program. But if I take the code that I've developed for a mental test run, I face a few problems. To see what I mean, picture yourself in the computer's shoes for a minute. (Well, a computer doesn't have shoes. Picture yourself in the computer's boots.)

You start at the top of the code shown in the previous section (the code that starts with the `Preliminary draft` comment). In the code's first statement, you display the words `Card Total`. So far, so good. But then you encounter the `while` loop and test the condition `total < 21`. Well, is `total` less than 21, or isn't it? Honestly, I'm tempted to make up an answer because I'm embarrassed about not knowing what the `total` variable's value is. (I'm sure the computer is embarrassed, too.)

The variable `total` must have a known value before the computer reaches the top of the `while` loop. Because a player starts with no cards at all, the initial `total` value should be 0. That settles it. I declare `int total = 0` at the top of the program.

But what about my friend, the `card` variable? Should I set `card` to zero also? No. There's no zero-valued card in a deck (at least, not when I'm playing fair). Besides, `card` should get a new value several times during the program's run.

Wait! In the previous sentence, the phrase *several times* tickles a neuron in my brain. It stimulates the *inside a loop* reflex. So I place an assignment to the `card` variable inside my `while` loop:

```
//This is a DRAFT - still NOT ready for prime time:
int card, total = 0;

System.out.println("Card    Total");

while (total < 21) {
    card = myRandom.nextInt(10) + 1;

    System.out.print(card);
    System.out.print("       ");
    System.out.println(total);
}

if (total == 21) {
    System.out.println("You win :-)");
} else {
    System.out.println("You lose :-(");
}
```

The code still has an error, and I can probably find the error with more computer role-playing. But instead, I get daring. I run this beta code to see what happens. Figure 12-8 shows part of a run.

```
Card    Total
5       0
10       0
3       0
4       0
8       0
5       0
5       0
1       0
6       0
7       0
2       0
1       0
3       0
4       0
8       0
3       0
8       0
```

Figure 12-8: An incorrect run.

Unfortunately, the run in Figure 12-8 doesn't stop on its own. This kind of processing is called an *infinite loop*. The loop runs and runs until someone trips over the computer's extension cord.

You can stop a program's run dead in its tracks. Look for the tiny red rectangle at the top of Eclipse's Console view. When you hover over the rectangle, the tooltip says "Terminate." When you click the rectangle, the active Java program stops running, and the rectangle turns gray.

From infinity to affinity

For some problems, an infinite loop is normal and desirable. Consider, for example, a real-time mission-critical application — air traffic control, or the monitoring of a heart-lung machine. In these situations, a program should run and run and run.

But a game of Twenty-One should end pretty quickly. In Figure 12-8, the game doesn't end because the `total` never reaches 21 or higher. In fact, the `total` is always zero. The problem is that my code has no statement to change the `total` variable's value. I should add each card's value to the `total`:

```
total += card;
```

Again, I ask myself where this statement belongs in the code. How many times should the computer execute this assignment statement? Once at the start of the program? Once at the end of the run? Repeatedly?

The computer should repeatedly add a card's value to the running total. In fact, the computer should add to the total each time a card gets drawn. So the preceding assignment statement should be inside the `while` loop, right alongside the statement that gets a new `card` value:

```
card = myRandom.nextInt(10) + 1;
total += card;
```

With this revelation, I'm ready to see the complete program. The code is in Listing 12-2, and two runs of the code are shown in Figure 12-5.

Listing 12-2: A Simplified Version of the Game Twenty-One

```
import java.util.Random;

class PlayTwentyOne {

    public static void main(String args[]) {
```

```
        Random myRandom = new Random();
        int card, total = 0;

        System.out.println("Card    Total");

        while (total < 21) {
            card = myRandom.nextInt(10) + 1;
            total += card;

            System.out.print(card);
            System.out.print("       ");
            System.out.println(total);
        }

        if (total == 21) {
            System.out.println("You win :-)");
        } else {
            System.out.println("You lose :-(");
        }
    }
}
```

If you've read this whole section, then you're probably exhausted. Creating a loop can be a lot of work. Fortunately, the more you practice, the easier it becomes.

Escapism

You can use a neat trick to make a program's output line up correctly. In Figure 12-5, the numbers `8 8`, then `6 14` (and so on) are displayed. I want these numbers to be right under the heading words `Card` and `Total`. Normally, you can get perfect vertical columns by pressing the tab key, but a computer program creates the output in Figure 12-5. How can you get a computer program to press the tab key?

In Java, there's a way. You can put `\t` inside double quote marks.

```
System.out.println
          ("Card\tTotal");

System.out.print(card);
System.out.print("\t");
System.out.println(total);
```

In the first statement, the computer displays `Card`, then jumps to the next tab stop on the screen, and then displays `Total`. In the next three statements, the computer displays a card number (like the number `6`), then jumps to the next tab stop (directly under the word `Total`), and then displays a `total` value (like the number `14`).

The `\t` combination of characters is an example of an *escape sequence*. Another of Java's escape sequences, the combination `\n`, moves the cursor to a new line. In other words, `System.out.print("Hello\n")` does the same thing as `System.out.println("Hello")`.

Thinking about Loops (Priming)

I remember when I was a young boy. We lived on Front Street in Philadelphia, near where the El train turned onto Kensington Avenue. Come early morning, I'd have to go outside and get water from the well. I'd pump several times before any water would come out. Ma and Pa called it "priming the pump."

These days I don't prime pumps. I prime `while` loops. Consider the case of a busy network administrator. She needs a program that extracts a username from an e-mail address. For example, the program reads

```
John@BurdBrain.com
```

and writes

```
John
```

How does the program do it? Like other examples in the chapter, this problem involves repetition:

```
Repeatedly do the following:
    Read a character.
    Write the character.
```

The program then stops the repetition when it finds the @ sign. I take a stab at writing this program. My first attempt doesn't work, but it's a darn good start. It's in Listing 12-3.

Listing 12-3: Trying to Get a Username from an E-Mail Address

```
/*
 * This code does NOT work, but I'm not discouraged.
 */
import java.util.Scanner;

class FirstAttempt {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        char symbol = ' ';

        while (symbol != '@') {
            symbol = myScanner.findWithinHorizon(".",0)
                                                .charAt(0);
            System.out.print(symbol);
        }
```

```
            System.out.println();
        }
}
```

When you run the code in Listing 12-3, you get the output shown in Figure 12-9. The user types one character after another — the letter `J`, then `o`, then `h`, and so on. At first, the program in Listing 12-3 does nothing. (The computer doesn't send any of the user's input to the program until the user presses Enter.) After the user types a whole e-mail address and presses Enter, the program gets its first character (the `J` in `John`).

Unfortunately, the program's output isn't what you expect. Instead of just the user name `John`, you get the username and the @ sign.

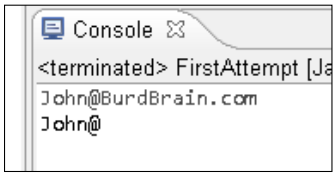


Figure 12-9: Oops! Got the @ sign, too.

To find out why this happens, follow the computer's actions as it reads the input `John@BurdBrain.com`:

```
Set symbol to ' ' (a blank space).

Is that blank space the same as an @ sign?
No, so perform a loop iteration.
    Input the letter 'J'.
    Print the letter 'J'.

Is that 'J' the same as an @ sign?
No, so perform a loop iteration.
    Input the letter 'o'.
    Print the letter 'o'.

Is that 'o' the same as an @ sign?
No, so perform a loop iteration.
    Input the letter 'h'.
    Print the letter 'h'.

Is that 'h' the same as an @ sign?
No, so perform a loop iteration.
    Input the letter 'n'.
    Print the letter 'n'.
```

```
Is that 'n' the same as an @ sign?  //Here's the problem.
No, so perform a loop iteration.
    Input the @ sign.
    Print the @ sign.               //Oops!

Is that @ sign the same as an @ sign?
Yes, so stop iterating.
```

Near the end of the program's run, the computer compares the letter `n` with the @ sign. Because `n` isn't an @ sign, the computer dives right into the loop:

- The first statement in the loop reads an @ sign from the keyboard.
- The second statement in the loop doesn't check to see if it's time to stop printing. Instead, that second statement just marches ahead and displays the @ sign.

After you've displayed the @ sign, there's no going back. You can't change your mind and undisplay the @ sign. So the code in Listing 12-3 doesn't quite work.

Working on the problem

You learn from your mistakes. The problem with Listing 12-3 is that, between reading and writing a character, the program doesn't check for an @ sign. Instead of doing "Test, Input, Print," it should do "Input, Test, Print." That is, instead of doing this:

```
Is that 'o' the same as an @ sign?
No, so perform a loop iteration.
    Input the letter 'h'.
    Print the letter 'h'.

Is that 'h' the same as an @ sign?
No, so perform a loop iteration.
    Input the letter 'n'.
    Print the letter 'n'.

Is that 'n' the same as an @ sign?  //Here's the problem.
No, so perform a loop iteration.
    Input the @ sign.
    Print the @ sign.               //Oops!
```

the program should do this:

```
    Input the letter 'o'.
Is that 'o' the same as an @ sign?
No, so perform a loop iteration.
    Print the letter 'o'.

    Input the letter 'n'.
Is that 'n' the same as an @ sign?
No, so perform a loop iteration.
    Print the letter 'n'.

    Input the @ sign.
Is that @ sign the same as an @ sign?
Yes, so stop iterating.
```

This cycle is shown in Figure 12-10.

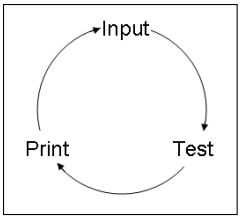


Figure 12-10: What the program needs to do.

You can try to imitate the following informal pattern:

```
    Input a character.
Is that character the same as an @ sign?
If not, perform a loop iteration.
    Print the character.
```

The problem is, you can't put a `while` loop's test in the middle of the loop:

```
//This code doesn't work
//  the way you want it to work:
{
   symbol = myScanner.findWithinHorizon(".",0).charAt(0);
while (symbol != '@')
   System.out.print(symbol);
}
```

You can't sandwich a `while` statement's condition between two of the statements that you intend to repeat. So what can you do? You need to follow the flow in Figure 12-11. Because every `while` loop starts with a test, that's where you jump into the circle, First Test, then Print, and finally Input.

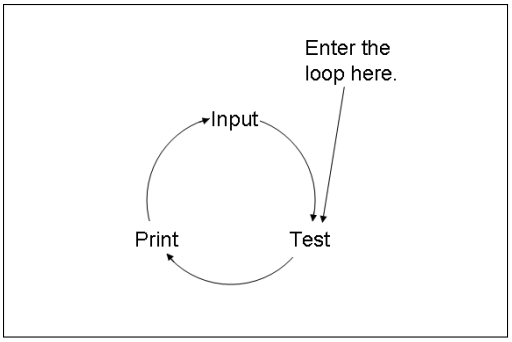


Figure 12-11: Jumping into a loop.

Listing 12-4 shows the embodiment of this "test, then print, then input" strategy.

Listing 12-4: Nice Try, But . . .

```
/*
 * This code almost works, but there's one tiny error:
 */
import java.util.Scanner;

class SecondAttempt {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        char symbol = ' ';

        while (symbol != '@') {
            System.out.print(symbol);
            symbol = myScanner.findWithinHorizon(".",0)
                                                .charAt(0);
        }

        System.out.println();
    }
}
```

A run of the Listing 12-4 code is shown in Figure 12-12. The code is almost correct, but I still have a slight problem. Notice the blank space before the user's input. The program races prematurely into the loop. The first time the computer executes the statements

```
System.out.print(symbol);
symbol = myScanner.findWithinHorizon(".",0).charAt(0);
```

the computer displays an unwanted blank space. Then the computer gets the `J` in `John`. In some applications, an extra blank space is no big deal. But in other applications, extra output can be disastrous.

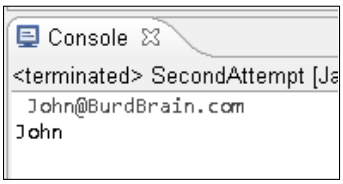


Figure 12-12: The computer displays an extra blank space.

Fixing the problem

Disastrous or not, an unwanted blank space is the symptom of a logical flaw. The program shouldn't display results before it has any meaningful results to display. The solution to this problem is called . . . (drumroll, please) . . . *priming the loop*. You pump `findWithinHorizon(".",0).charAt(0)` once to get some values flowing. Listing 12-5 shows you how to do it.

Listing 12-5 follows the strategy shown in Figure 12-13. First you get a character (the letter `J` in `John`, for example), and then you enter the loop. After you're in the loop, you test the letter against the @ sign and print the letter if it's appropriate to do so. Figure 12-14 shows a beautiful run of the `GetUserName` program.

Listing 12-5: How to Prime a Loop

```
/*
 * This code works correctly!
 */
import java.util.Scanner;

class GetUserName {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        char symbol;

        symbol = myScanner.findWithinHorizon(".",0)
                                              .charAt(0);

        while (symbol != '@') {
            System.out.print(symbol);
```

(continued)

Listing 12-5 ***(continued)***

```
            symbol = myScanner.findWithinHorizon(".",0)
                                                .charAt(0);
        }

        System.out.println();
    }
}
```

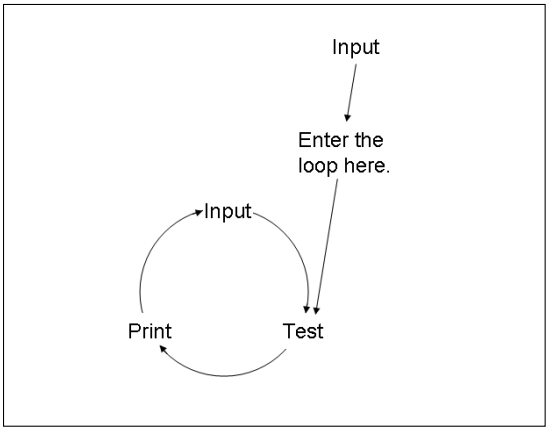


Figure 12-13: The strategy in Listing 12-5.

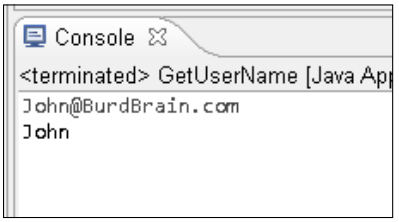


Figure 12-14: A run of the code in Listing 12-5.

The priming of loops in an important programming technique. But it's not the end of the story. In Chapters 14, 15, and 16, you can read about some other useful looping tricks.

Chapter 13

Piles of Files: Dealing with Information Overload

In This Chapter

- Using data on your hard drive
- Writing code to access the hard drive
- Troubleshooting input/output behavior

Consider these scenarios:

- You're a business owner with hundreds of invoices. To avoid boxes full of paper, you store invoice data in a file on your hard drive. You need customized code to sort and classify the invoices.
- You're an astronomer with data from scans of the night sky. When you're ready to analyze a chunk of data, you load the chunk onto your computer's hard drive.
- You're the author of a popular self-help book. Last year's fad was called the Self Mirroring Method. This year's craze is the Make Your Cake System. You can't modify your manuscript without converting to the publisher's new specifications. You need software to make the task bearable.

Each situation calls for a new computer program, and each program reads from a large data file. On top of all that, each program creates a brand new file containing bright, shiny results.

In previous chapters, the examples get input from the keyboard and send output to the Eclipse Console view. That's fine for small tasks, but you can't have the computer prompt you for each bit of night sky data. For big problems, you need lots of data, and the best place to store the data is on a computer's hard drive.

Running a Disk-Oriented Program

To deal with volumes of data, you need tools for reading from (and writing to) disk files. At the mere mention of disk files, some peoples' hearts start to palpitate with fear. After all, a disk file is elusive and invisible. It's stored somewhere inside your computer, with some magic magnetic process.

The truth is, getting data from a disk is very much like getting data from the keyboard. And printing data to a disk is like printing data to the computer screen.

In this book, displaying a program's text output "on the computer screen" means displaying text in Eclipse's Console view. If you shun Eclipse in favor of a different IDE (such as NetBeans or IntelliJ IDEA) or you shun all IDEs in favor of your system's command window, then, for you, "on the computer screen" means something slightly different. Please read between the lines as necessary. Also, I'm well aware that some computers have flash memory with no honest-to-goodness disks inside them. So terms like "disk-oriented" and "disk files" are showing signs of age. But let's face facts: A "record store" no longer sells vinyl records, and in U.S. measurement units, 12 inches are no longer the length the of the king's foot. Today's LCD screens no longer need saving. And, unlike the old mechanical car radios, a web page's radio buttons don't mark your favorite stations.

Consider the scenario when you run the code in the previous chapters. You type some stuff on the keyboard. The program takes this stuff and spits out some stuff of its own. The program sends this new stuff to the Console view. In effect, the flow of data goes from the keyboard, to the computer's innards, and on to the screen, as shown in Figure 13-1.

Of course, the goal in this chapter is the scenario in Figure 13-2. There's a file containing data on your hard drive. The program takes data from the disk file and spits out some brand-new data. The program then sends the new data to another file on the hard drive. In effect, the flow of data goes from a disk file, to the computer's innards, and on to another disk file.

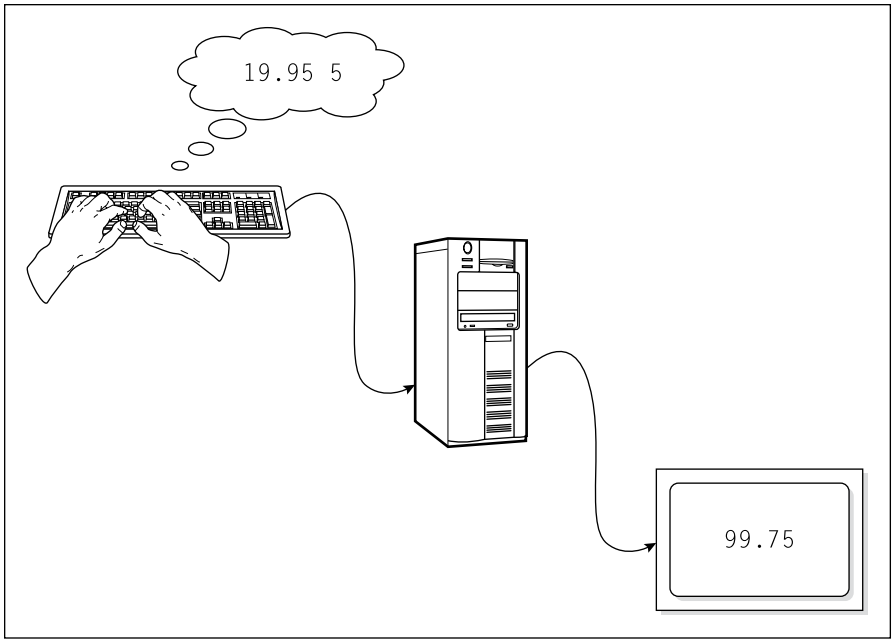


Figure 13-1: Using the keyboard and screen.

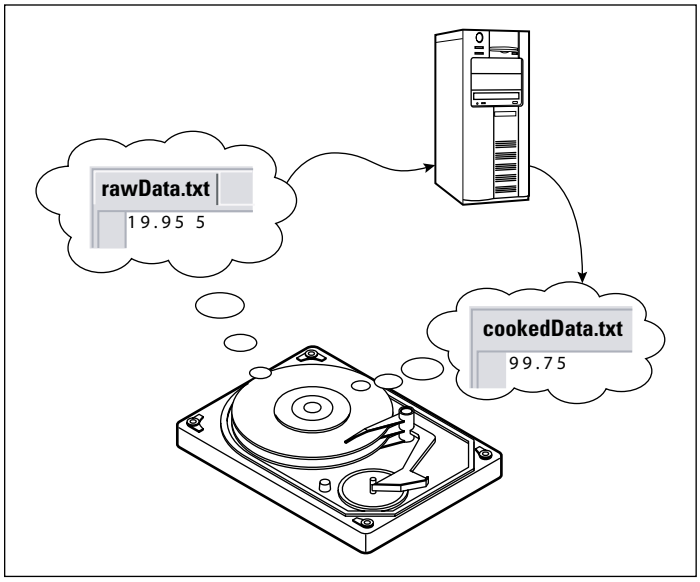


Figure 13-2: Using disk files.

The two scenarios in Figures 13-1 and 13-2 are very similar. In fact, it helps to remember these fundamental points:

- **The stuff in a disk file is no different from the stuff that you type on a keyboard.**

  If a keyboard-reading program expects you to type `19.95 5`, then the corresponding disk-reading program expects a file containing those same characters, `19.95 5`. If a keyboard-reading program expects you to press Enter and type more characters, then the corresponding disk-reading program expects more characters on the next line in the file.

- **The stuff in a disk file is no different from the stuff that you see in Eclipse's Console view.**

  If a screen-printing program displays the number `99.75`, then the corresponding disk-writing program writes the number `99.75` to a file. If a screen-printing program moves the cursor to the next line, then the corresponding disk-writing program creates a new line in the file.

If you have trouble imagining what you have in a disk file, just imagine the text that you would type on the keyboard or the text that you would see on the computer screen (that is, in Eclipse's Console view). That same text can appear in a file on your disk.

A sample program

Listing 13-1 contains a keyboard/screen program. The program multiplies unit price by quantity to get a total price. A run of the code is shown in Figure 13-3.

Listing 13-1: Using the Keyboard and the Screen

```
import java.util.Scanner;

class ComputeTotal {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        double unitPrice, quantity, total;

        unitPrice = myScanner.nextDouble();
        quantity = myScanner.nextInt();

        total = unitPrice * quantity;

        System.out.println(total);
    }
}
```

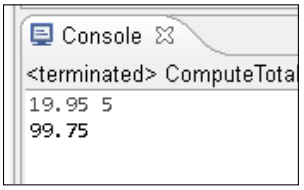


Figure 13-3: Read from the keyboard; write to the screen.

The goal is to write a program like the one in Listing 13-1. But instead of talking to your keyboard and screen, this new program talks to your hard drive. The new program reads unit price and quantity from your hard drive, and writes the total back to your hard drive.

Java's API has everything you need for interacting with a hard drive. A nice example is in Listing 13-2.

Listing 13-2: Using Input and Output Files

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;

class ReadAndWrite {

    public static void main(String args[])
                        throws FileNotFoundException {

        Scanner diskScanner =
            new Scanner(new File("rawData.txt"));
        PrintStream diskWriter =
            new PrintStream("cookedData.txt");
        double unitPrice, quantity, total;

        unitPrice = diskScanner.nextDouble();
        quantity = diskScanner.nextInt();

        total = unitPrice * quantity;

        diskWriter.println(total);
    }
}
```

For a guide to the care and feeding of the `rawData.txt` file (whose name appears in Listing 13-2), see the upcoming "Creating an input file" section.

Creating code that messes with your hard drive

"I _____ (print your name)___ agree to pay $_____ each month on the ___th day of the month."

Fill in the blanks. That's all you have to do. Reading input from a disk can work the same way. Just fill in the blanks in Listing 13-3.

Listing 13-3: A Template to Read Data from a Disk File

```
/*
 * Before Eclipse can compile this code,
 * you must fill in the blanks.
 */
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

class ____________ {

    public static void main(String args[])
                        throws FileNotFoundException {

        Scanner diskScanner =
            new Scanner(new File("__________"));


        _______ = diskScanner.nextInt();
        _______ = diskScanner.nextDouble();
        _______ = diskScanner.nextLine();
        _______ = diskScanner.findWithinHorizon(".",0)
                                              .charAt(0);

        // Etc.
    }
}
```

To use Listing 13-3, make up a name for your class. Insert that name into the first blank space. Type the name of the input file in the second space (between the quotation marks). Then, to read a whole number from the input file, call `diskScanner.nextInt`. To read a number that has a decimal point, call `diskScanner.nextDouble`. You can call any of the `Scanner` methods in Chapter 5's Table 5-1 (the same methods you call when you get keystrokes from the keyboard).

The stuff in Listing 13-3 isn't a complete program. Instead, it's a *code template* — a half-baked piece of code, with spaces for you to fill in.

With the template in Listing 13-3, you can input data from a disk file. With a similar template, you can write output to a file. The template is in Listing 13-4.

Listing 13-4: A Template to Write Data to a Disk File

```
/*
 * Before Eclipse can compile this code,
 * you must fill in the blanks.
 */
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;

class ___________ {

    public static void main(String args[])
                        throws FileNotFoundException {

        PrintStream diskWriter =
                         new PrintStream("__________");

        diskWriter.print(_____);
        diskWriter.println(_____);

        // Etc.
    }
}
```

To use Listing 13-4, insert the name of your class into the first blank space. Type the name of the output file in the space between the quotation marks. Then, to write part of a line to the output file, call `diskWriter.print`. To write the remainder of a line to the output file, call `diskWriter.println`.

The Eclipse has a built-in feature for creating and inserting code templates. To get started using Eclipse templates, choose Window➪Preferences (in Windows) or Eclipse➪Preferences (on a Mac). In the resulting Preferences dialog box, choose Java➪Editor➪Templates. Creating new templates isn't simple. But if you poke around a bit, you accomplish a lot.

If your program gets input from one disk file and writes output to another, then combine the stuff from Listings 13-3 and 13-4. When you do, you get a program like the one in Listing 13-2.

A quick look at Java's disk access facilities

Templates like the ones in Listings 13-3 and 13-4 are very nice. But knowing how the templates work is even better. Here are a few tidbits describing the inner workings of Java's disk access code.

- **A `PrintStream` is something you can use for writing output.**

  A `PrintStream` is like a `Scanner`. The big difference is, a `Scanner` is for reading input and a `PrintStream` is for writing output. To see what I mean, look at Listing 13-2. Notice the similarity between the statements that use `Scanner` and the statements that use `PrintStream`.

  The word `PrintStream` is defined in the Java API.

- **In Listing 13-2, the expression `new File("rawData.txt")` plays the same role that `System.in` plays in so many other programs.**

  Just as `System.in` stands for the computer's keyboard, the expression `new File("rawData.txt")` stands for a file on your computer's hard drive. When the computer calls `new File("rawData.txt")`, the computer creates something like `System.in` — something you can stuff inside the `new Scanner( )` parentheses.

  The word `File` is defined in the Java API.

- **A `FileNotFoundException` is something that may go wrong during an attempt to read input from a disk file (or an attempt to write output to a disk file).**

  Disk file access is loaded with pitfalls. Even the best programs run into disk access trouble occasionally. So to brace against such pitfalls, Java insists on your adding some extra words to your code.

  In Listing 13-2, the added words `throws FileNotFoundException` form a *throws clause*. A throws clause is a kind of disclaimer. Putting a throws clause in your code is like saying, "I realize that this code can run into trouble."

  Of course, in the legal realm, you often have no choice about signing disclaimers. "If you don't sign this disclaimer, then the surgeon won't operate on you." Okay, then I'll sign it. The same is true with a Java throws clause. If you put things like `new PrintStream("cookedData.txt")` in your code, and you don't add something like `throws FileNotFoundException`, then the Java compiler refuses to compile your code.

  So when do you need this `throws FileNotFoundException` clause, and when should you do without it? Well, having certain things in your code — things like `new PrintStream("cookedData.txt")` — force you to create a throws clause. You can learn all about the kinds of things that demand throws clauses. But at this point, it's better to concentrate on other programming issues. As a beginning Java programmer, the safest thing to do is to follow the templates in Listings 13-3 and 13-4.

  The word `FileNotFoundException` is . . . you guessed it . . . defined in the Java API.

- **To create this chapter's code, I made up the names `diskScanner` and `diskWriter`.**

  The words `diskScanner` and `diskWriter` don't come from the Java API. In place of `diskScanner` and `diskWriter`, you can use any names you want. All you have to do is to use the names consistently within each of your Java programs.

Running the sample program

Testing the code in Listing 13-2 is a three-step process. Here's an outline of the three steps:

1. **Create the `rawData.txt` file.**
2. **Run the code in Listing 13-2.**
3. **View the contents of the `cookedData.txt` file.**

The next few sections cover each step in detail.

Creating an input file

You can use any plain old text editor to create an input file for the code in Listing 13-2. In this section, I show you how to use Eclipse's built-in editor.

To create an input file:

1. **Select a project in Eclipse's Package Explorer.**

   In this example, select the 13-02 project (the project containing the code from Listing 13-2).

   In the Package Explorer, select a branch whose label is the name of a project. (Select the `13-02` branch to run the code in Listing 13-2.) Don't select an item within a project. (For example, don't select the `src` branch or the `(default package)` branch.)

2. **In Eclipse's main menu, choose File➪New➪File.**

   Eclipse's New File dialog box opens.

3. **In the File Name field, type the name of your new data file.**

   You can type any name that your computer considers to be a valid file name. For this section's example, I used the name `rawData.txt`, but other names, such as `rawData.dat`, `rawData`, or `raw123.01.dataFile`, are fine. I try to avoid troublesome names (including short, uninformative names and names containing blank spaces), but the name you choose is entirely up to you (and your computer's operating system, and your boss's whims, and your customer's specifications).

4. **Click Finish.**

   The file's name appears in Eclipse's Package Explorer. An empty editor (with the new file's name on its tab) appears in Eclipse's editor area.

5. **Type text in the editor.**

   To create this section's example, I typed the text `19.95 5`, as shown in Figure 13-4. To create your own example, type whatever text your program needs during its run.

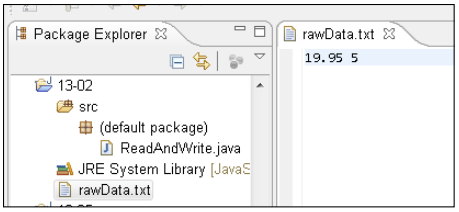


Figure 13-4: Editing an input file.

Running the code

To run the code, do the same thing you do with any other Java program. Select the project you want to run (project 13-02 in this example). Then choose Run➪Run As➪Java Application.

When you run the program in Listing 13-2, no text appears in Eclipse's Console view. This total lack of any noticeable output gives some people the willies. The truth is, a program like the one in Listing 13-2 does all its work behind the scenes. The program has no statements that read from the keyboard and has no statements that print to the screen. So if you have a very loud hard drive, you may hear a little chirping sound when you choose Run➪Run As➪Java Application, but you won't type any program input, and you won't see any program output.

The program sends all its output to a file on your hard drive. So what do you do to see the file's contents?

Viewing the output file

To see the output of the program in Listing 13-2, follow these steps:

1. **In the Project Explorer, select the `13-02` project branch.**
2. **In the main menu, choose File➪Refresh.**
3. **In the Project Explorer, expand the `13-02` project branch.**

   A new file named `cookedData.txt` appears in the Package Explorer tree (in the 13-02 project).

4. **Double-click the `cookedData.txt` branch in the Package Explorer tree.**

   The contents of `cookedData.txt` appear in an Eclipse editor (see Figure 13-5).

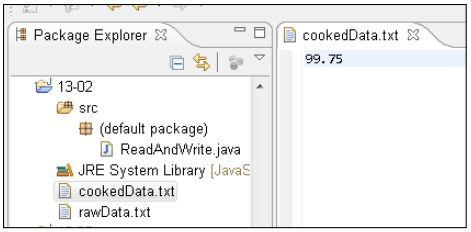


Figure 13-5: Viewing an output file.

Troubleshooting problems with disk files

When you run the code in Listing 13-2, the computer executes `new Scanner (new File("rawData.txt"))`. If the Java virtual machine can't find the `rawData.txt` file, then you see a message like the one shown in Figure 13-6. This error message can be very frustrating. In many cases, you know darn well that there's a `rawData.txt` file on your hard drive. The stupid computer simply can't find it.

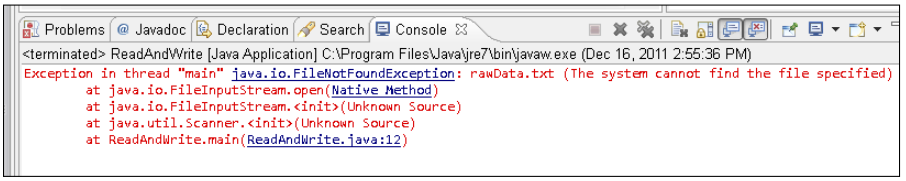


Figure 13-6: The computer can't find your file.

There's no quick, surefire way to fix this problem. But you should always check the following things first:

- **Check again for a file named `rawData.txt`.**

  Open My Computer (on Windows) or Finder (on a Mac) and poke around for a file with that name.

  The filenames displayed in My Computer and Finder can be misleading. You may see the name `rawData` even though the file's real name is `rawData.txt`. To fix this problem once and for all, see Chapter 2.

- **Check the spelling of the file's name.**

  Make sure that the name in your program is exactly the same as the name of the file on your hard drive. Just one misplaced letter can keep the computer from finding a file.

- **If you use Linux (or a flavor of UNIX other than Mac OS X), check the capitalization of the file's name.**

  In Linux, and in many versions of UNIX, the difference between uppercase and lowercase can baffle the computer.

- **Check that the file is in the correct directory.**

  Sure, you have a file named `rawData.txt`. But don't expect your Java program to look in every folder on your hard drive to find the file. How do you know which folder should house files like `rawData.txt`?

  Here's how it works: Each Eclipse project has its own folder on your computer's hard drive. You see the 13-02 project folder and its src subfolder in Figure 13-5. But in Figure 13-7, Windows Explorer shows the 13-02 folder, its src subfolder, and its other subfolders named .`settings` and `bin`. (Mac users can see the same subfolders in a Finder window.)

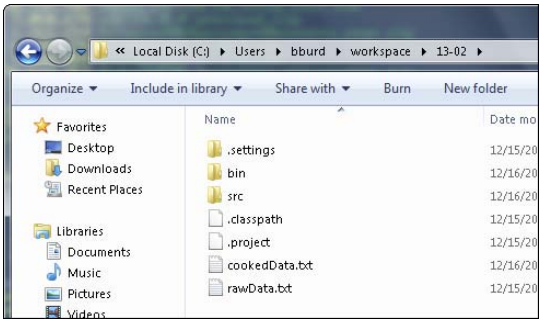


Figure 13-7: The contents of the 13-02 project folder on your computer's hard drive.

The `src`, `bin` and .`settings` folders contain files of their own. But in Figure 13-7, the `rawData.txt` and `cookedData.txt` files are immediately inside the 13-02 project folder. In other words, the `rawData.txt` and `cookedData.txt` files live in the root of the 13-02 project folder.

When you run this section's example, the `rawData.txt` file should be in the root of the 13-02 project folder on your hard drive. That's why, in Step 1 of the "Creating an input file" section, I remind you to select the 13-02 project folder and not the project's `src` subfolder.

Figure 13-7 shows input and output files in the root of their Eclipse project. But in general, file locations can be tricky, especially if you switch from Eclipse to an unfamiliar IDE. The general rule (about putting input and output files immediately inside a project directory) may not apply in other programming environments.

So here's a trick you can use: Whatever IDE you use (or even if you create Java programs without an IDE), run this stripped-down version of the code in Listing 13-2:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;

class JustWrite {

    public static void main(String args[])
        throws FileNotFoundException {

        PrintStream diskWriter =
            new PrintStream("cookedData.txt");
        diskWriter.println(99.75);
    }
}
```

This program has no need for a stinking `rawData.txt` file. If you run this code and get no error messages, then search your hard drive for this program's output (the `cookedData.txt` file). Note the name of the folder that contains the `cookedData.txt` file. When you put `rawData.txt` in this same folder, then any problem you had running the Listing 13-2 code should go away.

Check the `rawData.txt` file's content.

It never hurts to peek inside the `rawData.txt` file and make sure that the file contains the numbers `19.95 5`. If `rawData.txt` doesn't appear in Eclipse's editor area, find the Listing 13-2 project (the project named 13-02) in the Package Explorer. Double-clicking the project's `rawData.txt` branch makes that file appear in Eclipse's editor area.

By default, Java's `Scanner` class looks for blank spaces between input values. So this example's `rawData.txt` file should contain `19.95 5`, not `19.955` and not `19.95,5`.

The `Scanner` class looks for any kind of *whitespace* between the values. These whitespace characters may include blank spaces, tabs, and newlines. So, for example, the `rawData.txt` file may contain `19.95  5` (with several blank spaces between `19.95` and `5`), or may have `19.95` and `5` on two separate lines.

Writing a Disk-Oriented Program

Listing 13-2 is very much like Listing 13-1. In fact, you can go from Listing 13-1 to Listing 13-2 with some simple editing. Here's how:

- **Add the following import declarations to the beginning of your code:**

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;
```

- **Add the following throws clause to the method header:**

```
throws FileNotFoundException
```

- **In the call to `new Scanner`, replace `System.in` with a call to `new File` as follows:**

```
Scanner aVariableName =
    new Scanner(new File("inputFileName"))
```

- **Create a `PrintStream` for writing output to a disk file:**

```
PrintStream anotherVariableName =
    new PrintStream("outputFileName");
```

- **Use the `Scanner` variable name in calls to `nextInt`, `nextLine`, and so on.**

  For example, to go from Listing 13-1 to Listing 13-2, I change

```
unitPrice = myScanner.nextDouble();
quantity = myScanner.nextInt();
```

  to

```
unitPrice = diskScanner.nextDouble();
quantity = diskScanner.nextInt();
```

- **Use the `PrintStream` variable name in calls to `print` and `println`.**

  For example, to go from Listing 13-1 to Listing 13-2, I change

```
System.out.println(total);
```

  to

```
diskWriter.println(total);
```

Reading from a file

All the `Scanner` methods can read from existing disk files. For example, to read a word from a file named `mySpeech`, use code of the following kind:

```
Scanner diskScanner =
    new Scanner(new File("mySpeech"));

String oneWord = diskScanner.next();
```

To read a character from a file named `letters.dat` and then display the character on the screen, you can do something like this:

```
Scanner diskScanner =
    new Scanner(new File("letters.dat"));

System.out.println(
    diskScanner.findWithinHorizon(".",0).charAt(0));
```

Notice how I read from a file named `mySpeech`, not `mySpeech.txt` or `mySpeech.doc`. Anything that you put after the dot is called a *filename extension*, and for a file full of numbers and other data, the filename extension is optional. Sure, a Java program must be called *`something`*`.java`, but a data file can be named `mySpeech.txt`, `mySpeech.reallymine.allmine`, or just `mySpeech`. As long as the name in your `new File` call is the same as the filename on your computer's hard drive, everything is okay.

Writing to a file

The `print` and `println` methods can write to disk files. Here are some examples:

- During a run of the code in Listing 13-2, the variable `total` stores the number 99.75. To deposit 99.75 into the `cookedData.txt` file, you execute

  ```
  diskWriter.println(total);
  ```

  This `println` call writes to a disk file because of the following line in Listing 13-2:

  ```
  PrintStream diskWriter =
      new PrintStream("cookedData.txt");
  ```

- In another version of the program, you may decide not to use a `total` variable. To write 99.75 to the `cookedData.txt` file, you can call

  ```
  diskWriter.println(unitPrice * quantity);
  ```

- To display `OK` on the screen, you can make the following method call:

```
System.out.print("OK");
```

  To write `OK` to a file named `approval.txt`, you can use the following code:

```
PrintStream diskWriter =
    new PrintStream("approval.txt");

diskWriter.print("OK");
```

- You may decide to write `OK` as two separate characters. To write to the screen, you can make the following calls:

```
System.out.print('O');
System.out.print('K');
```

  And to write `OK` to the `approval.txt` file, you can use the following code:

```
PrintStream diskWriter =
    new PrintStream("approval.txt");

diskWriter.print('O');
diskWriter.print('K');
```

- Like their counterparts for `System.out`, the disk-writing `print` and `println` methods differ in their end-of-line behaviors. For example, you want to display the following text on the screen:

```
Hankees  Socks
7        3
```

  To do this, you can make the following method calls:

```
System.out.print("Hankees  ");
System.out.println("Socks");
System.out.print(7);
System.out.print("        ");
System.out.println(3);
```

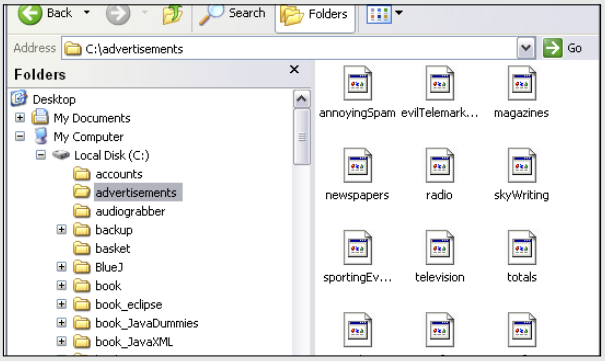
  To plant the same text into a file named `scores.dat`, you can use the following code:

```
PrintStream diskWriter =
    new PrintStream("scores.dat");

diskWriter.print("Hankees  ");
diskWriter.println("Socks");
diskWriter.print(7);
diskWriter.print("        ");
diskWriter.println(3);
```

Name that file

What if a file that contains data is not in your program's project folder? If that's the case, when you call `new File`, the file's name must include folder names. For example, in Windows, your `TallyBudget.java` program might be in your `c:\Users\MyUserName\workspace\13-09` folder, and a file named `totals` might be in a folder named `c:\advertisements`. (See the following figure.)



Then, to refer to the `totals` file, you include the folder name, the filename, and (to be on the safe side) the drive letter:

```
Scanner diskScanner = new Scanner
(new File("c:\\advertisements\\totals"));
```

Notice that I use double backslashes to separate the drive letter, the folder name, and the filename. To find out why, look at the sidebar entitled "Escapism" in Chapter 12. The string `"\totals"` with a single backslash stands for a tab, followed by `otals`. But in this example, the file's name is `totals`, not `otals`. With a single backslash, the name `...advertisements\totals"` would not work correctly.

Inside quotation marks, you use the double backslash to indicate what would usually be a single backslash. So the string `"c:\\advertisements\\totals"` stands for `c:\advertisements\totals`. That's good because `c:\advertisements\totals` is the way you normally refer to a file in Windows.

If you want to sidestep all this backslash confusion, you can use forward slashes to specify each file's location. Windows responds exactly the same way to `new File("c:\\advertisements\\totals")` and to `new File("c:/advertisements/totals")`. And if you use UNIX, Linux, or a Macintosh, the double backslash nonsense doesn't apply to you. Just write

```
Scanner diskScanner =
 new Scanner (new File(
 "/Users/me/advertisements/totals"));
```

or something similar that reflects your system's directory structure.

Writing, Rewriting, and Re-rewriting

Given my mischievous ways, I tried a little experiment. I asked myself what would happen if I ran the same file-writing program more than once. So I created a tiny program (the program in Listing 13-5), and I ran the program twice. Then I examined the program's output file. The output file (shown in Figure 13-8) contains only two letters.

Listing 13-5: A Little Experiment

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;

class WriteOK {

    public static void main(String args[])
        throws FileNotFoundException {

        PrintStream diskWriter =
            new PrintStream(new File("approval.txt"));

        diskWriter.print  ('O');
        diskWriter.println('K');
    }
}
```

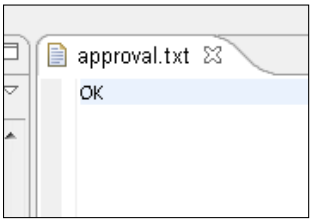


Figure 13-8: Testing the waters.

Here's the sequence of events from the start to the end of the experiment:

1. **Before I run the code in Listing 13-5, my computer's hard drive has no `approval.txt` file.**

   That's okay. Every experiment has to start somewhere.

2. **I run the code in Listing 13-5.**

   The call to `new PrintStream` in Listing 13-5 creates a file named `approval.txt`. Initially, the new `approval.txt` file contains no characters. Later in Listing 13-5, calls to `print` and `println` put characters

in the file. So after running the code, the `approval.txt` file contains two letters: the letters `OK`.

3. **I run the code from Listing 13-5 a second time.**

   At this point, I could imagine seeing `OKOK` in the `approval.txt` file. But that's not what I see in Figure 13-8. After running the code twice, the `approval.txt` file contains just one `OK`. Here's why:

   - The call to `new PrintStream` in Listing 13-5 deletes my existing `approval.txt` file. The call creates a new, empty `approval.txt` file.
   - After creating a new `approval.txt` file, the `print` method call drops the letter `O` into the new file.
   - The `println` method call adds the letter `K` to the same `approval.txt` file.

So that's the story. Each time you run the program, it trashes whatever `approval.txt` file is already on the hard drive. Then the program adds data to a newly created `approval.txt` file.

Chapter 14

Creating Loops within Loops

In This Chapter

- Analyzing loop strategies
- Diagnosing loop problems
- Creating nested loops

If you're an editor at Wiley Publishing, please don't read the next few paragraphs. In the next few paragraphs, I give away an important trade secret (something you really don't want me to do).

I'm about to describe a surefire process for writing a best-selling *For Dummies* book. Here's the process:

Write several words to create a sentence. Do this several times to create a paragraph.

```
Repeat the following to form a paragraph:
   Repeat the following to form a sentence:
      Write a word.
```

Repeat the previous instructions several times to make a section. Make several sections and then make several chapters.

```
Repeat the following to form a best-selling book
in the For Dummies series:
   Repeat the following to form a chapter:
      Repeat the following to form a section:
         Repeat the following to form a paragraph:
            Repeat the following to form a sentence:
               Write a word.
```

This process involves a loop within a loop within a loop within a loop within a loop. It's like a verbal M.C. Escher print. Is it useful, or is it frivolous?

Well, in the world of computer programming, this kind of thing happens all the time. Most five-layered loops are hidden behind method calls, but two-layered loops within loops are everyday occurrences. So this chapter tells you how to compose a loop within a loop. It's very useful stuff.

By the way, if you're a Wiley Publishing editor, you can start reading again from this point onward.

Paying Your Old Code a Little Visit

The program in Listing 12-5 extracts a username from an e-mail address. For example, the program reads

```
John@BurdBrain.com
```

from the keyboard, and writes

```
John
```

to the screen. Let me tell you . . . in this book, I have some pretty lame excuses for writing programs, but this simple e-mail example tops the list! Why would you want to type something on the keyboard, only to have the computer display part of what you typed? There must be a better use for code of this kind.

Sure enough, there is. The BurdBrain.com network administrator has a list of 10,000 employees' e-mail addresses. More precisely, the administrator's hard drive has a file named `email.txt`. This file contains 10,000 e-mail addresses, with one address on each line, as shown in Figure 14-1.

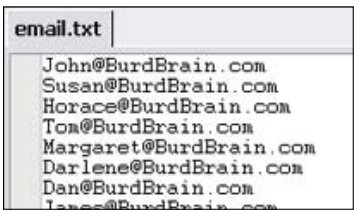


Figure 14-1: A list of e-mail addresses.

The company's e-mail software has an interesting feature. To send e-mail within the company, you don't need to type an entire e-mail address. For example, to send e-mail to John, you can type the username `John` instead of `John@BurdBrain.com`. (This `@BurdBrain.com` part is called the *host name*.)

So the company's network administrator wants to distill the content of the `email.txt` file. She wants a new file, `usernames.txt`, that contains usernames with no host names, as shown in Figure 14-2.

Figure 14-2: Usernames extracted from the list of e-mail addresses.

Reworking some existing code

To solve the administrator's problem, you need to modify the code in Listing 12-5. The new version gets an e-mail address from a disk file and writes a username to another disk file. The new version is in Listing 14-1.

Listing 14-1: From One File to Another

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;

class ListOneUsername {

   public static void main(String args[])
                        throws FileNotFoundException {

      Scanner diskScanner =
         new Scanner(new File("email.txt"));
      PrintStream diskWriter =
         new PrintStream("usernames.txt");
      char symbol;

      symbol =
```

(continued)

Listing 14-1 *(continued)*

```
            diskScanner.findWithinHorizon(".",0).charAt(0);

        while (symbol != '@') {
            diskWriter.print(symbol);
            symbol =
              diskScanner.findWithinHorizon(".",0).charAt(0);
        }

        diskWriter.println();
    }
}
```

Listing 14-1 does almost the same thing as its forerunner in Listing 12-5. The only difference is that the code in Listing 14-1 doesn't interact with the user. Instead, the code in Listing 14-1 interacts with disk files.

Running your code

Here's how you run the code in Listing 14-1:

1. **Create a file named `email.txt` in your Eclipse project directory.**

   In the `email.txt` file, put just one e-mail address. Any address will do, as long as the address contains an @ sign.

2. **Put the `ListOneUsername.java` file (the code from Listing 14-1) in your project's `src/(default package)` directory.**

3. **Run the code in Listing 14-1.**

   When you run the code, you see nothing interesting in the Console view. What a pity!

4. **View the contents of the `usernames.txt` file.**

   If your `email.txt` file contains `John@BurdBrain.com`, then the `usernames.txt` file contains `John`.

For more details on any of these steps, see the discussion accompanying Listings 13-2, 13-3, and 13-4 in Chapter 13.

Creating Useful Code

The previous section describes a network administrator's problem — creating a file filled with usernames from a file filled with e-mail addresses. The code in Listing 14-1 solves part of the problem — it extracts just one e-mail

address. That's a good start, but to get just one username, you don't need a computer program. A pencil and paper does the trick.

So don't keep the network administrator waiting any longer. In this section, you develop a program that processes dozens, hundreds, and even thousands of e-mail addresses from a file on your hard drive.

First, you need a strategy to create the program. Take the statements in Listing 14-1 and run them over and over again. Better yet, have the statements run themselves over and over again. Fortunately, you already know how to do something over and over again: You use a loop. (See Chapter 12 for the basics on loops.)

So here's the strategy: Take the statements in Listing 14-1 and enclose them in a larger loop:

```
while (not at the end of the email.txt file) {
    Execute the statements in Listing 14-1
}
```

Looking back at the code in Listing 14-1, you see that the statements in that code have a `while` loop of their own. So this strategy involves putting one loop inside another loop:

```
while (not at the end of the email.txt file) {
    //Blah-blah

    while (symbol != '@') {
        //Blah-blah-blah
    }

    //Blah-blah-blah-blah
}
```

Because one loop is inside the other, they're called *nested loops*. The old loop (the `symbol != '@'` loop) is the *inner loop*. The new loop (the end-of-file loop) is called the *outer loop*.

Checking for the end of a file

Now all you need is a way to test the loop's condition. How do you know when you're at the end of the `email.txt` file?

The answer comes from Java's `Scanner` class. This class's `hasNext` method answers `true` or `false` to the following question:

> Does the `email.txt` file have anything to read in it (beyond what you've already read)?

If the program's `findWithinHorizon` calls haven't gobbled up all the characters in the `email.txt` file, then the value of `diskScanner.hasNext()` is `true`. So to keep looping while you're not at the end of the `email.txt` file, you do the following:

```
while (diskScanner.hasNext()) {
    Execute the statements in Listing 14-1
}
```

The first realization of this strategy is in Listing 14-2.

Listing 14-2: The Mechanical Combining of Two Loops

```
/*
 * This code does NOT work (but
 * you learn from your mistakes).
 */

import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;

class ListAllUsernames {

   public static void main(String args[])
                        throws FileNotFoundException {

      Scanner diskScanner =
         new Scanner(new File("email.txt"));
      PrintStream diskWriter =
         new PrintStream("usernames.txt");
      char symbol;

      while (diskScanner.hasNext()) {
         symbol = diskScanner.findWithinHorizon(".",0)
                                              .charAt(0);

         while (symbol != '@') {
            diskWriter.print(symbol);
            symbol = diskScanner.findWithinHorizon(".",0)
                                               .charAt(0);
         }

         diskWriter.println();
      }
   }
}
```

When you run the code in Listing 14-2, you get the disappointing response shown in Figure 14-3.

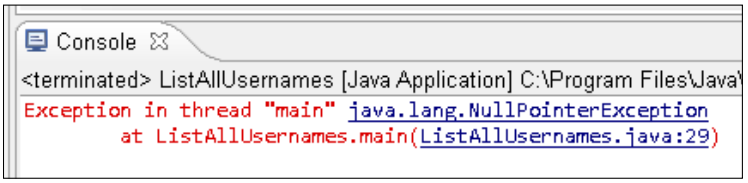


Figure 14-3: You goofed.

How it feels to be a computer

What's wrong with the code in Listing 14-2? To find out, I role-play the computer. "If I were a computer, what would I do when I execute the code in Listing 14-2?"

The first several things that I'd do are pictured in Figure 14-4. I would read the `J` in `John`, then write the `J` in `John`, and then read the letter `o` (also in `John`).

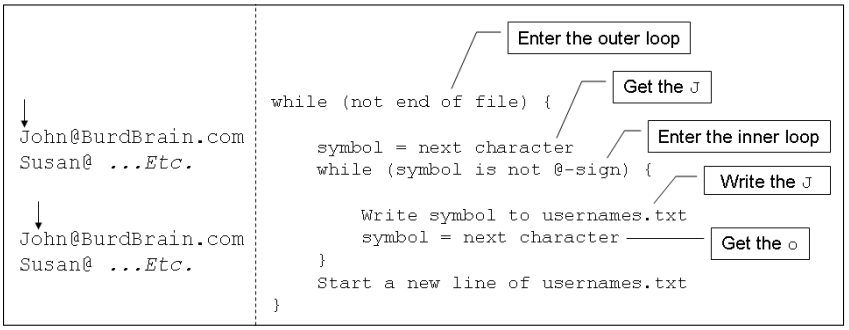


Figure 14-4: Role-playing the code in Listing 14-2.

After a few trips through the inner loop, I'd get the @ sign in `John@BurdBrain.com`, as shown in Figure 14-5.

Finding this @ sign would jump me out of the inner loop and back to the top of the outer loop, as shown in Figure 14-6.

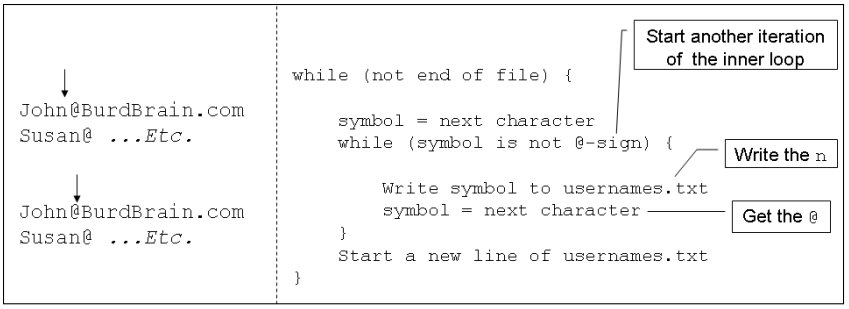


Figure 14-5: Reaching the end of the username.

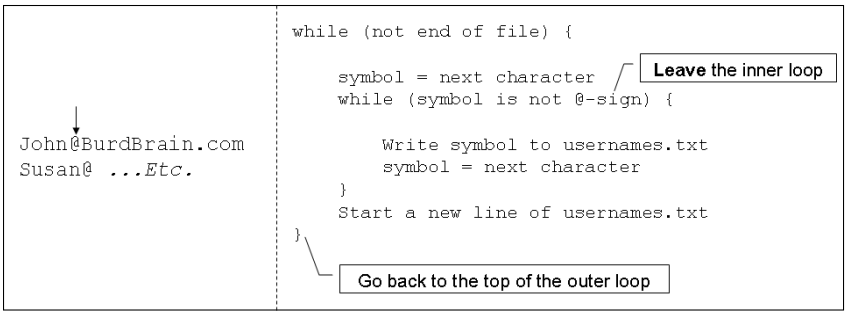

John@BurdBrain.com
Susan@ ...Etc.
while (not end of file) {
symbol = next character
while (symbol is not @-sign) {
Write symbol to usernames.txt
symbol = next character
}
Start a new line of usernames.txt
}
Leave the inner loop
Go back to the top of the outer loop

Figure 14-6: Leaving the inner loop.

I'd get the `B` in `BurdBrain`, and sail back into the inner loop. But then (horror of horrors!) I'd write that `B` to the `usernames.txt` file (see Figure 14-7).

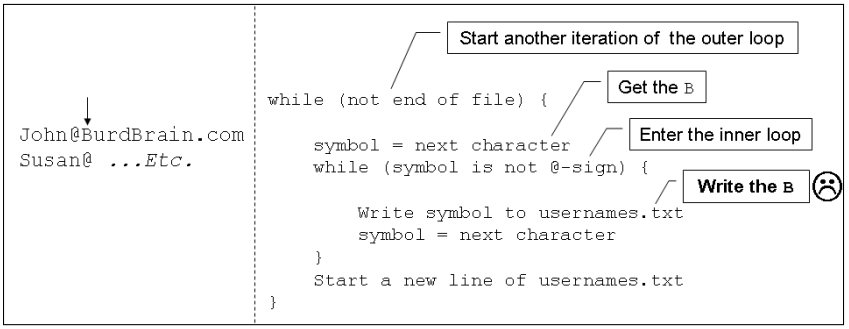


Figure 14-7: The error of my ways.

There's the error! You don't want to write host names to the `usernames.txt` file. When the computer found the @ sign, it should have skipped past the rest of John's e-mail address.

At this point, you have a choice. You can jump straight to the corrected code in Listing 14-3, or you can read on to find out about the error message in Figure 14-3.

Why the computer accidentally pushes past the end of the file

Ah! You're wondering why Figure 14-3 has that nasty error message.

Once again, I role-play the computer. I've completed the steps in Figure 14-7. I shouldn't process `BurdBrain.com` with the inner loop. But unfortunately, I do.

I keep running and processing more e-mail addresses. When I get to the end of the last e-mail address, I grab the `m` in `BurdBrain.com` and go back to test for an @ sign, as shown in Figure 14-8.

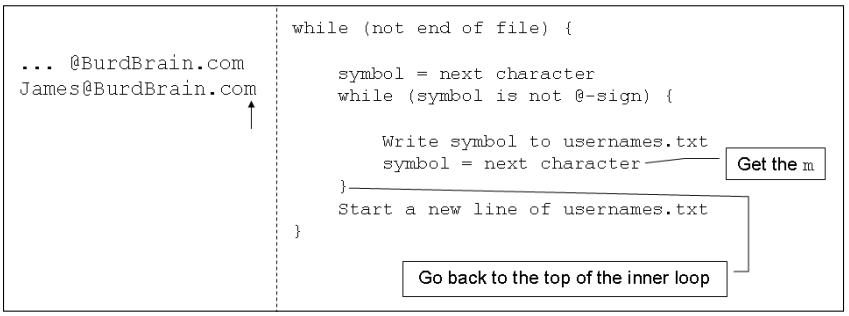


Figure 14-8: The journey's last leg.

Now I'm in trouble. This last `m` certainly isn't an @ sign. So I jump into the inner loop and try to get yet another character (see Figure 14-9). The `email.txt` file has no more characters, so Java sends an error message to the computer screen. (The `NullPointerException` error message is back in Figure 14-3.)

Solving the problem

Listing 14-3 has the solution to the problem described with Figures 14-1 and 14-2. The code in this listing is almost identical to the code in Listing 14-2. The only difference is the added call to `nextLine`. When the computer reaches an @ sign, this `nextLine` call gobbles up the rest of the input line.

(In other words, the `nextLine` call gobbles up the rest of the e-mail address. The idea works because each e-mail address is on its own separate line.) After chewing and swallowing `@BurdBrain.com`, the computer moves gracefully to the next line of input.

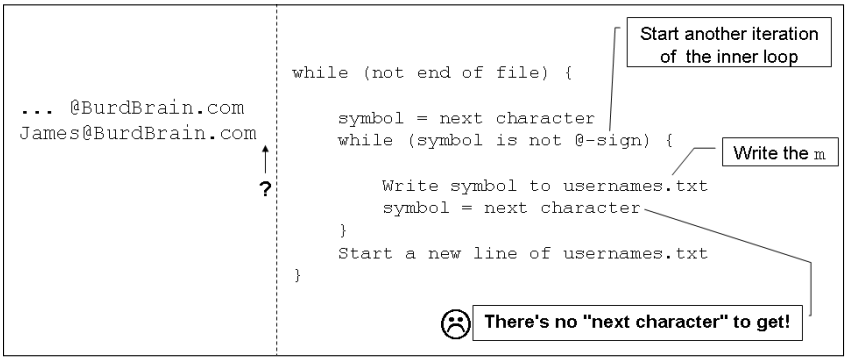


Figure 14-9: Trying to read past the end of the file.

Listing 14-3: That's Much Better!

```
/*
 * This code is correct!!
 */

import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;

class ListAllUsernames {

   public static void main(String args[])
                        throws FileNotFoundException {

      Scanner diskScanner =
         new Scanner(new File("email.txt"));
      PrintStream diskWriter =
         new PrintStream("usernames.txt");
      char symbol;
```

```
        while (diskScanner.hasNext()) {
            symbol = diskScanner.findWithinHorizon(".",0)
                                                  .charAt(0);

            while (symbol != '@') {
                diskWriter.print(symbol);
                symbol = diskScanner.findWithinHorizon(".",0)
                                                   .charAt(0);
            }

            diskScanner.nextLine();
            diskWriter.println();
        }
    }
}
```

To run the code in Listing 14-3, you need an `email.txt` file — a file like the one shown in Figure 14-1. In the `email.txt` file, type several e-mail addresses. Any addresses will do, as long as each address contains an @ sign and each address is on its own separate line. Save the `email.txt` file in your project directory along with the `ListAllUsernames.java` file (the code from Listing 14-3). For more details, see the discussion accompanying Listings 13-2, 13-3, and 13-4 in Chapter 13.

With Listing 14-3, you've reached an important milestone. You've analyzed a delicate programming problem and found a complete, working solution. The tools you used included thinking about strategies and role-playing the computer. As time goes on, you can use these tools to solve bigger and better problems.

Chapter 15

The Old Runaround

In This Chapter

- Creating repetitive actions
- Creating loops within loops
- Insisting on a valid response from the user
- Looping through enumerated values

I remember it distinctly — the sense of dread I would feel on the way to Aunt Edna's house. She was a kind old woman, and her intentions were good. But visits to her house were always so agonizing.

First, we'd sit in the living room and talk about other relatives. That was okay, as long as I understood what people were talking about. Sometimes, the gossip would be about adult topics, and I'd become very bored.

After all the family chatter, my father would help Aunt Edna with her bills. That was fun to watch because Aunt Edna had a genetically inherited family ailment. Like me and many of my ancestors, Aunt Edna couldn't keep track of paperwork to save her life. It was as if the paper had allergens that made Aunt Edna's skin crawl. After ten minutes of useful bill paying, my father would find a mistake, an improper tally, or something else in the ledger that needed attention. He'd ask Aunt Edna about it, and she'd shrug her shoulders. He'd become agitated trying to track down the problem, while Aunt Edna rolled her eyes and smiled with ignorant satisfaction. It was great entertainment.

Then, when the bill paying was done, we'd sit down to eat dinner. That's when I would remember why I dreaded these visits. Dinner was unbearable. Aunt Edna believed in Fletcherism — a health movement whose followers chewed each mouthful of food 100 times. The more devoted followers used a chart, with a different number for the mastication of each kind of food. The minimal number of chews for any food was 32 — one chomp for each tooth in your mouth. People who did this said they were "Fletcherizing."

Mom and Dad thought the whole Fletcher business was silly, but they respected Aunt Edna and felt that people her age should be humored, not defied. As for me, I thought I'd explode from the monotony. Each meal lasted forever. Each mouthful was an ordeal. I can still remember my mantra — the words I'd say to myself without meaning to do so:

```
I've chewed 0 times so far.
Have I chewed 100 times yet? If not, then
    Chew!
    Add 1 to the number of times that I've chewed.
    Go back to "Have I chewed"
        to find out if I'm done yet.
```

Repeating Statements a Certain Number Times (Java for Statements)

Life is filled with examples of counting loops. And computer programming mirrors life (. . . or is it the other way around?). When you tell a computer what to do, you're often telling the computer to print three lines, process ten accounts, dial a million phone numbers, or whatever. Because counting loops are so common in programming, the people who create programming languages have developed statements just for loops of this kind. In Java, the statement that repeats something a certain number of times is called a *for* statement. An example of a `for` statement is in Listing 15-1.

Listing 15-1: Horace Fletcher's Revenge

```
import static java.lang.System.out;

class AuntEdnaSettlesForTen {

    public static void main(String args[]) {

        for (int count = 0; count < 10; count++) {
            out.print("I've chewed ");
            out.print(count);
            out.println(" time(s).");
        }

        out.println("10 times! Hooray!");
        out.println("I can swallow!");
    }
}
```

Figure 15-1 shows you what you get when you run the program of Listing 15-1:

- The `for` statement in Listing 15-1 starts by setting the `count` variable equal to 0.
- Then the `for` statement tests to make sure that `count` is less than 10 (which it certainly is).
- Then the `for` statement dives ahead and executes the printing statements between the curly braces. At this early stage of the game, the computer prints `I've chewed 0 time(s).`
- Then the `for` statement executes `count++` — that last thing inside the `for` statement's parentheses. This last action adds 1 to the value of `count`.

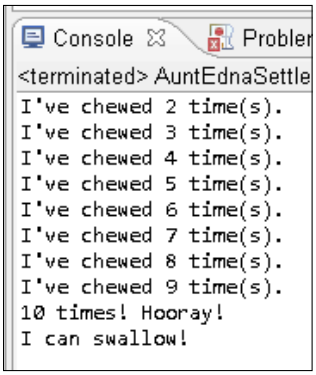


Figure 15-1: Chewing ten times.

This ends the first iteration of the `for` statement in Listing 15-1. Of course, this loop has more to it than just one iteration:

- With `count` now equal to 1, the `for` statement checks again to make sure that `count` is less than 10. (Yes, 1 is smaller than 10.)
- Because the test turns out okay, the `for` statement marches back into the curly braced statements and prints `I've chewed 1 time(s)` on the screen.
- Then the `for` statement executes that last `count++` inside its parentheses. The statement adds 1 to the value of `count`, increasing the value of `count` to 2.

And so on. This whole thing keeps being repeated over and over again until, after ten iterations, the value of `count` finally reaches 10. When this happens, the check for `count` being less than 10 fails, and the loop's execution ends. The computer jumps to whatever statement comes immediately after the `for` statement. In Listing 15-1, the computer prints `10 times! Hooray! I can swallow!` The whole process is illustrated in Figure 15-2.

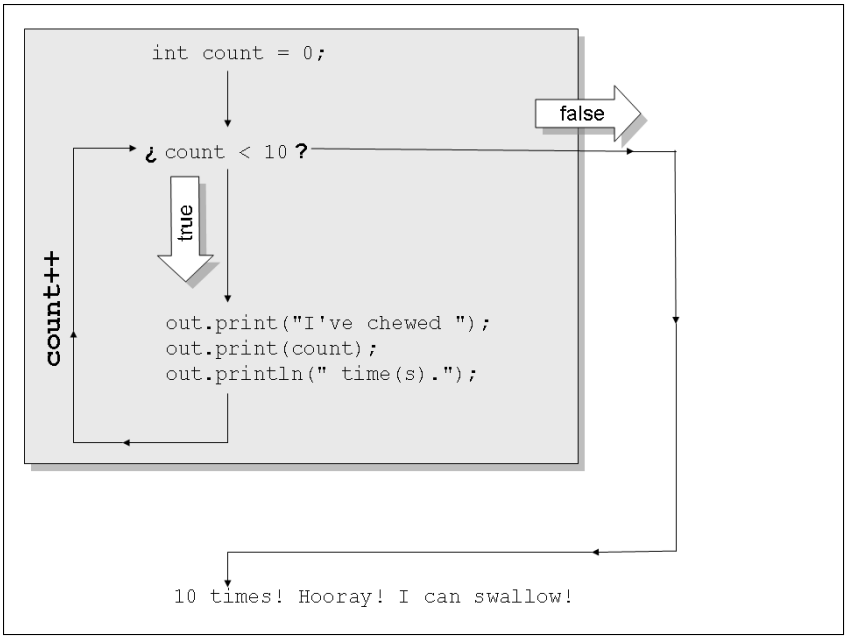


Figure 15-2: The action of the for loop in Listing 15-1.

The anatomy of a for statement

A typical `for` statement looks like this:

```
for (Initialization; Condition; Update) {
    Statements
}
```

After the word `for`, you put three things in parentheses: an *Initialization,* a *Condition,* and an *Update.*

Each of the three items in parentheses plays its own distinct role:

- **Initialization:** The initialization is executed once, when the run of your program first reaches the `for` statement.
- **Condition:** The condition is tested several times (at the start of each iteration).
- **Update:** The update is also evaluated several times (at the end of each iteration).

If it helps, think of the loop as if its text is shifted all around:

```
//This is NOT real code
int count = 0
for count < 0 {
    out.print("I've chewed ");
    out.print(count);
    out.println(" time(s).");
    count++;
}
```

You can't write a real `for` statement this way. (The compiler would throw code like this right into the garbage can.) Even so, this is the order in which the parts of the `for` statement are executed.

The first line of a `for` statement (the word `for` followed by stuff in parentheses) isn't a complete statement. So you almost never put a semicolon after the stuff in parentheses. If you make a mistake and type a semicolon,

```
// DON'T DO THIS:
for (int count = 0; count < 10; count++); {
```

you usually put the computer into an endless, do-nothing loop. The computer's cursor just sits there and blinks until you forcibly stop the program's run.

Initializing a for loop

Look at the first line of the `for` loop in Listing 15-1 and notice the declaration `int count = 0`. That's something new. When you create a `for` loop, you can declare a variable (like `count`) as part of the loop initialization.

If you declare a variable in the initialization of a `for` loop, you can't use that variable outside the loop. For example, in Listing 15-1, try putting `out.println(count)` after the end of the loop:

```
//This code does not compile.
for (int count = 0; count < 10; count++) {
    out.print("I've chewed ");
    out.print(count);
    out.println(" time(s).");
}

out.print(count);   //The count variable doesn't
                    //  exist here.
```

With this extra reference to the `count` variable, the compiler gives you an error message. You can see the message in Figure 15-3. If you're not experienced with `for` statements, the message may surprise you. "Whadaya mean 'count cannot be resolved to a variable'? There's a `count` variable declaration just four lines above that statement." Ah, yes. But the `count` variable is declared in the `for` loop's initialization. Outside the `for` loop, that `count` variable doesn't exist.

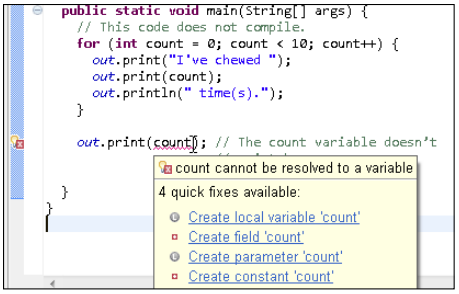

public static void main(String[] args) {
// This code does not compile.
for (int count = 0; count < 10; count++) {
out.print("I've chewed ");
out.print(count);
out.println(" time(s).");
}
out.print(count); // The count variable doesn't t
count cannot be resolved to a variable
4 quick fixes available:
Create local variable 'count'
Create field 'count'
Create parameter 'count'
Create constant 'count'

Figure 15-3: What count variable? I don't see a count variable.

To use a variable outside of a `for` statement, you have to declare that variable outside the `for` statement. You can even do this with the `for` statement's counting variable. Listing 15-2 has an example.

Listing 15-2: Using a Variable Declared Outside of a for Loop

```
import static java.lang.System.out;

class AuntEdnaDoesItAgain {

    public static void main(String args[]) {
        int count;

        for (count = 0; count < 10; count++) {
            out.print("I've chewed ");
            out.print(count);
            out.println(" time(s).");
        }

        out.print(count);
        out.println(" times! Hooray!");
        out.println("I can swallow!");
    }
}
```

A run of the code in Listing 15-2 looks exactly like the run for Listing 15-1. The run is pictured in Figure 15-1. Unlike its predecessor, Listing 15-2 enjoys the luxury of using the `count` variable to display the number 10. It can do this because in Listing 15-2, the `count` variable belongs to the entire `main` method, and not to the `for` loop alone.

Versatile looping statements

If you were stuck on a desert island with only one kind of loop, what kind would you want to have? The answer is, you can get along with any kind of loop. The choice between a `while` loop and a `for` loop is about the code's style and efficiency. It's not about necessity.

Anything that you can do with a `for` loop, you can do with a `while` loop as well. Consider, for example, the `for` loop in Listing 15-1. Here's how you can achieve the same effect with a `while` loop:

```
int count = 0;
while (count < 10) {
  out.print("I've chewed ");
  out.print(count);
  out.println(" time(s).");
  count++;
}
```

In the `while` loop, you have explicit statements to declare, initialize, and increment the `count` variable.

The same kind of trick works in reverse. Anything that you can do with a `while` loop, you can do with a `for` loop as well. But turning certain `while` loops into `for` loops seems strained and unnatural. Consider a `while` loop from Listing 12-2:

```
while (total < 21) {
  card =
    myRandom.nextInt(10) + 1;
  total += card;
  System.out.print(card);
  System.out.print("     ");
  System.out.println(total);
}
```

Turning this loop into a `for` loop means wasting most of the stuff inside the `for` loop's parentheses:

```
for ( ; total < 21 ; ) {
  card =
    myRandom.nextInt(10) + 1;
  total += card;
  System.out.print(card);
  System.out.print("     ");
  System.out.println(total);
}
```

The preceding `for` loop has a condition, but it has no initialization and no update. That's okay. Without an initialization, nothing special happens when the computer first enters the `for` loop. And without an update, nothing special happens at the end of each iteration. It's strange, but it works.

Usually, when you write a `for` statement, you're counting how many times to repeat something. But, in truth, you can do just about any kind of repetition with a `for` statement.

Notice the words `for (count = 0` in Listing 15-2. Because `count` is declared before the `for` statement, you don't declare `count` again in the `for` statement's initialization. I tried declaring `count` twice, as in the following code:

```
//This does NOT work:
int count;

for (int count = 0; count < 10; count++) {
    ...etc.
```

and Eclipse told me to clean up my act:

```
Duplicate local variable count                    ^
```

Using Nested for Loops

Because you're reading *Beginning Programming with Java For Dummies,* 3rd Edition, I assume that you manage a big hotel. The next chapter tells you everything you need to know about hotel management. But before you begin reading that chapter, you can get a little preview in this section.

I happen to know that your hotel has 9 floors, and each floor of your hotel has 20 rooms. On this sunny afternoon, someone hands you a flash drive containing a file full of numbers. You copy this `hotelData` file to your hard drive and then display the file in Eclipse's editor. You see the stuff shown in Figure 15-4.

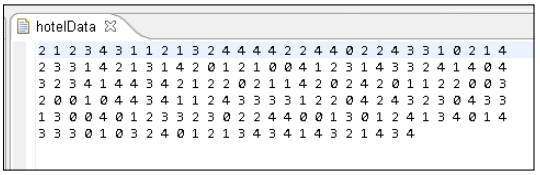


Figure 15-4: A file containing hotel occupancy data.

This file gives the number of guests in each room. For example, at the start of the file, you see `2 1 2`. This means that, on the first floor, Room 1 has 2 guests, Room 2 has 1 guest, and Room 3 has 2 guests. After reading 20 of these numbers, you see `0 2 2`. So, on the second floor, Room 1 has 0 guests, Room 2 has 2 guests, and Room 3 has 2 guests. The story continues until the last number in the file. According to that number, Room 20 on the ninth floor has 4 guests.

You'd like a more orderly display of these numbers — a display of the kind in Figure 15-5. So you whip out your keyboard to write a quick Java program.

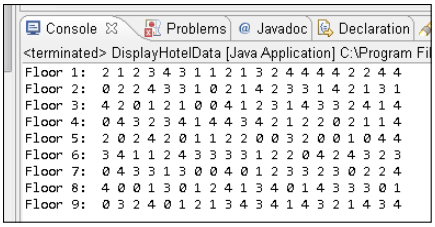


Figure 15-5: A readable display of the data in Figure 15-4.

As in some other examples, you decide which statements go where by asking yourself how many times each statement should be executed. For starters, the display in Figure 15-5 has 9 lines, and each line has 20 numbers:

```
for (each of 9 floors)
    for (each of 20 rooms on a floor)
        get a number from the file and
        display the number on the screen.
```

So your program has a `for` loop within a `for` loop — a pair of nested `for` loops.

Next, you notice how each line begins in Figure 15-5. Each line contains the word `Floor`, followed by the floor number. Because this `Floor` display occurs only 9 times in Figure 15-5, the statements to print this display belong in the *`for each of 9 floors`* loop (and not in the *`for each of 20 rooms`* loop). The statements should be before the *`for each of 20 rooms`* loop because this `Floor` display comes once before each line's 20-number display:

```
for (each of 9 floors)
    display "Floor" and the floor number,
    for (each of 20 rooms on a floor)
        get a number from the file and
        display the number on the screen.
```

You're almost ready to write the code. But there's one detail that's easy to forget. (Well, it's a detail that I always forget.) After displaying 20 numbers, the program advances to a new line. This new-line action happens only 9 times during the run of the program, and it always happens *after* the program displays 20 numbers:

```
for (each of 9 floors)
    display "Floor" and the floor number,
    for (each of 20 rooms on a floor)
        get a number from the file and
        display the number on the screen,
    Go to the next line.
```

That does it. That's all you need. The code to create the display of Figure 15-5 is in Listing 15-3.

Listing 15-3: Hey! Is This a For-by-For?

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
import static java.lang.System.out;

class DisplayHotelData {

   public static void main(String args[])
      throws FileNotFoundException {

      Scanner diskScanner =
          new Scanner(new File("hotelData"));

      for (int floor = 1; floor <= 9; floor++) {
         out.print("Floor ");
         out.print(floor);
         out.print(":  ");

         for (int roomNum = 1; roomNum <= 20; roomNum++) {
            out.print(diskScanner.nextInt());
            out.print(' ');
         }

         out.println();
      }
   }
}
```

The code in Listing 15-3 has the variable `floor` going from 1 to 9 and has the variable `roomNum` going from 1 to 20. Because the `roomNum` loop is inside the `floor` loop, the writing of 20 numbers happens 9 times. That's good. It's exactly what I want.

Repeating Until You Get What You Need (Java do Statements)

I introduce Java's `while` loop in Chapter 12. When you create a `while` loop, you write the loop's condition first. After the condition, you write the code that gets repeatedly executed.

```
while (Condition) {
    Code that gets repeatedly executed
}
```

This way of writing a `while` statement is no accident. The look of the statement emphasizes an important point — that the computer always checks the condition before executing any of the repeated code.

If the loop's condition is never true, then the stuff inside the loop is never executed — not even once. In fact, you can easily cook up a `while` loop whose statements are never executed (although I can't think of a reason why you would ever want to do it):

```
//This code doesn't print anything:
int twoPlusTwo = 2 + 2;
while (twoPlusTwo == 5) {
    System.out.println("Are you kidding?");
    System.out.println("2+2 doesn't equal 5.");
    System.out.print  ("Everyone knows that");
    System.out.println(" 2+2 equals 3.");
}
```

In spite of this silly `twoPlusTwo` example, the `while` statement turns out to be the most useful of Java's looping constructs. In particular, the `while` loop is good for situations in which you must look before you leap. For example: "While money is in my account, write a mortgage check every month." When you first encounter this statement, if your account has a zero balance, you don't want to write a mortgage check — not even one check.

But at times (not many), you want to leap before you look. In a situation when you're asking the user for a response, maybe the user's response makes sense, but maybe it doesn't. Maybe the user's finger slipped, or perhaps the user didn't understand the question. In many situations, it's important to correctly interpret the user's response. If the user's response doesn't make sense, you must ask again.

Getting a trustworthy response

Consider a program that deletes several files. Before deleting anything, the program asks for confirmation from the user. If the user types `Y`, then delete; if the user types `N`, then don't delete. Of course, deleting files is serious stuff. Mistaking a bad keystroke for a "yes" answer can delete the company's records. (And mistaking a bad keystroke for a "no" answer can preserve the company's incriminating evidence.) So if there's any doubt about the user's response, the program should ask the user to respond again.

Pause a moment to think about the flow of actions — what should and shouldn't happen when the computer executes the loop. A loop of this kind doesn't need to check anything before getting the user's first response. Indeed, before the user gives the first response, the loop has nothing to check. The loop shouldn't start with "as long as the user's response is invalid, get another response from the user." Instead, the loop should just leap ahead, get a response from the user and then check the response to see whether it made sense. The code to do all this is in Listing 15-4.

Listing 15-4: Repeat Before You Delete

```
/*
 * DISCLAIMER: Neither the author nor Wiley Publishing,
 * Inc., nor anyone else even remotely connected with the
 * creation of this book, assumes any responsibility
 * for any damage of any kind due to the use of this code,
 * or the use of any work derived from this code,
 * including any work created partially or in full by
 * the reader.
 *
 * Sign here:_____________________________________
 */

import java.util.Scanner;
import java.io.IOException;

class IHopeYouKnowWhatYoureDoing {

  public static void main(String args[])
      throws IOException {

    Scanner myScanner = new Scanner(System.in);
    char reply;

    do {
      System.out.print("Reply with Y or N...");
      System.out.print("  Delete all .keep files? ");
      reply =
        myScanner.findWithinHorizon(".", 0).charAt(0);
    } while (reply != 'Y' && reply != 'N');

    if (reply == 'Y') {

      String opSystem = System.getProperty("os.name");

      if (opSystem.contains("Windows")) {

        Runtime.getRuntime().exec("cmd /c del *.keep");

      } else if (opSystem.contains("Mac")) {
```

```
            Runtime.getRuntime().exec(
                new String[] { "/bin/sh", "-c",
                    "rm -f *.keep" });
        }
    }
  }
}
```

The `exec.("cmd ... )` code in Listing 15-4 works on all the industrial-strength versions of Microsoft Windows, including Windows 7 and Windows XP. To get the same effect in Windows 95, 98, or Me, you have to change the last line of code as follows:

```
Runtime.getRuntime().exec("start command /c del *.keep");
```

One way or another, the call to `Runtime.getRuntime...yada-yada` deletes all files whose names end with `.keep`.

Java has no easy way to refer to all the files whose names end in `.keep`. So in Listing 15-4, I call `Runtime.getRuntime().exec` to execute an operating system command. This `Runtime` business can be tricky to use, so don't fret over the details. Just take my word for it — the calls to `Runtime.getRuntime().exec` in Listing 15-4 delete files.

To delete only one file, use the Java API's `delete` method. For example, to delete your `myData.txt` file, execute `new File("myData.txt").delete()`. And don't let the `new File` part fool you. The code `new File("myData.txt").delete()` gets rid of a file that's already on your computer's hard drive.

The `Runtime.getRuntime().exec` method is one of those "you need a throws clause" methods that I introduce in Chapter 13. Unlike the methods in Chapter 13, the `exec` method forces you to throw an `IOException`. And with a `throws IOException` clause comes an `import java.io.IOException` declaration.

At this point, you may be wondering how I know that the `exec` method needs a `throws IOException` clause. How many other Java API methods require throws clauses, and how do you find out about all these things? The answer is, you can find this information in Java's API documentation. For details, see this book's website (`allmycode.com/BeginProg3`).

Deleting files

A run of the Listing 15-4 program is shown in Figure 15-6. Before deleting a bunch of files, the program asks the user if it's okay to do the deletion. If the

user gives one of the two expected answers (`Y` or `N`), then the program proceeds according to the user's wishes. But if the user enters any other letter (or any digit, punctuation symbol, or whatever), then the program asks the user for another response.

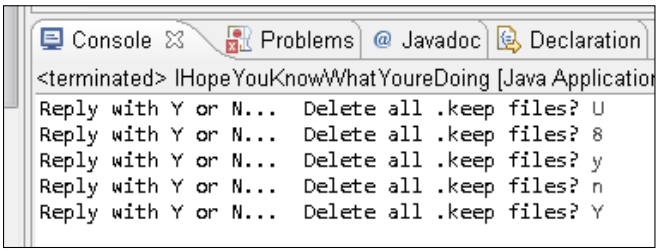

Console
Problems
Javadoc
Declaration
<terminated> IHopeYouKnowWhatYoureDoing [Java Applicatio
Reply with Y or N... Delete all .keep files? U
Reply with Y or N... Delete all .keep files? 8
Reply with Y or N... Delete all .keep files? y
Reply with Y or N... Delete all .keep files? n
Reply with Y or N... Delete all .keep files? Y

Figure 15-6: No! Don't do it!

In Figure 15-6, the user hems and haws for a while, first with the letter `U`, then the digit `8`, and then with lowercase letters. Finally, the user enters `Y`, and the program deletes the `.keep` files. If you compare the files on your hard drive (before and after the run of the program), you'll see that the program trashes files with names ending in `.keep`.

If you use Eclipse, here's how you can tell that files are being deleted:

1. **Create a Java project containing the code in Listing 15-4.**

   If you followed the steps in Chapter 2 for importing this book's examples, you can skip this create-a-project step and use the existing 15-04 project.

2. **In the Package Explorer, select the project.**

   Don't select any of the project's subfolders. (For example, don't select the project's `src` folder.) Instead, select the project's root. For more info about a project's root, see Chapter 13.

3. **In Eclipse's main menu, choose File➪New➪File.**

   Eclipse's New File dialog box appears.

   In the New File dialog box, make sure that the name of your project's root folder is in the box's Enter Or Select The Parent Folder field. For example, if you followed the steps in Chapter 2 for importing this book's examples, make sure that 15-04 (and no other text) appears in the Enter Or Select The Parent Folder field.

4. **In the dialog box's File Name field, type the name of your new data file.**

   Type `irreplaceableInfo.keep` or something like that.

5. **Click Finish.**

   The file's name appears in Eclipse's Package Explorer. For this experiment, you don't have to add any text to the file. The file exists only to be deleted.

6. **Repeat Steps 2 through 5 a few more times.**

   Create files named *somethingOrOther*`.keep` and files that don't have `.keep` in their names.

7. **Run the program.**

   When the program runs, type **Y** to delete the `.keep` files. (The program deletes `.keep` files only in this project's root directory. The program's run has no effect on any files outside of the root directory.)

   After running the program, you want to check to make sure that the program deleted the `.keep` files.

8. **In the Package Explorer, select the project's root (again, for good measure).**

9. **In Eclipse's main menu, choose File⇨Refresh.**

   Eclipse takes another look at the project directory and lists the directory's files in the Package Explorer's tree. Assuming that the program did its job correctly, files with names ending in `.keep` no longer appear in the tree.

Using Java's do statement

To write the program in Listing 15-4, you need a loop — a loop that repeatedly asks the user whether the `.keep` files should be deleted. The loop continues to ask until the user gives a meaningful response. The loop tests its condition at the end of each iteration, after each of the user's responses.

That's why the program in Listing 15-4 has a *do* loop (also known as a *do . . . while* loop). With a `do` loop, the program jumps right in, executes some statements, and then checks a condition. If the condition is true, then the program goes back to the top of the loop for another go-around. If the condition is false, then the computer leaves the loop (and jumps to whatever code comes immediately after the loop). The action of the loop in Listing 15-4 is illustrated in Figure 15-7.

A closer look at the do statement

The format of a `do` loop is

```
do {
   Statements
} while (Condition)
```

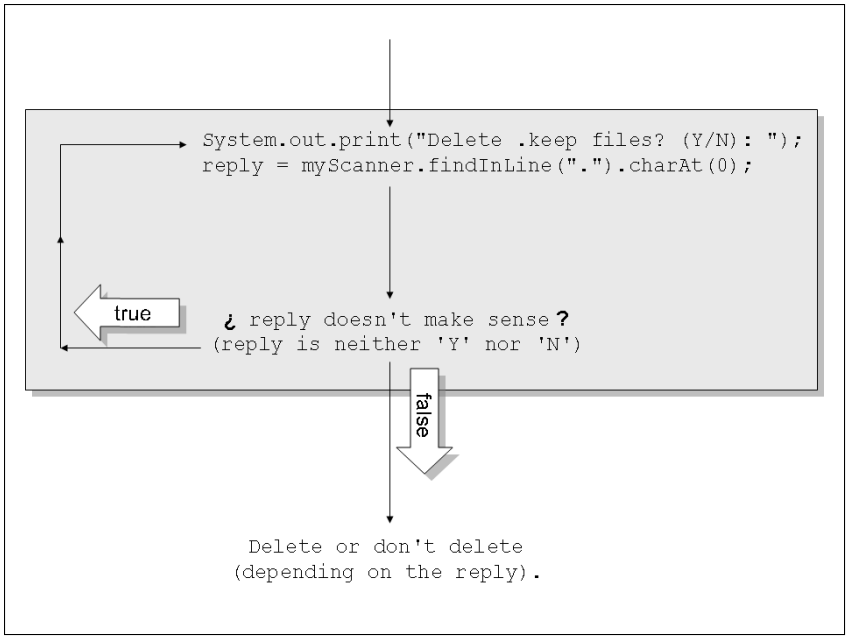


Figure 15-7: Here we go loop, do loop.

Writing the *Condition* at the end of the loop reminds me that the computer executes the *Statement* inside the loop first. After the computer executes the *Statement*, the computer goes on to check the *Condition*. If the *Condition* is true, the computer goes back for another iteration of the *Statement*.

With a `do` loop, the computer always executes the statements inside the loop at least once:

```
//This code prints something:
int twoPlusTwo = 2 + 2;
do {
    System.out.println("Are you kidding?");
    System.out.println("2+2 doesn't equal 5.");
    System.out.print  ("Everyone knows that");
    System.out.println(" 2+2 equals 3.");
} while (twoPlusTwo == 5);
```

This code displays `Are you kidding? 2+2 doesn't equal 5 . . .` *and so on* and then tests the condition `twoPlusTwo == 5`. Because `twoPlusTwo == 5` is false, the computer doesn't go back for another iteration. Instead, the computer jumps to whatever code comes immediately after the loop.

Repeating with Predetermined Values (Java's Enhanced for Statement)

Most people say that they "never win anything." Other people win raffles, drawings, and contests, but they don't win things. Well, I have news for these people — other people don't win things, either. Nobody wins things. That's how the laws of probability work. Your chance of winning one of the popular U.S. lottery jackpots is roughly 1 in 135,000,000. If you sell your quarter-million dollar house and use all the money to buy lottery tickets, your chance of winning is still only 1 in 540. If you play every day of the month (selling a house each day), your chance of winning the jackpot is still less than 1 in 15.

Of course, nothing in the previous paragraph applies to me. I don't buy lottery tickets, but I often win things. My winning streak started a few years ago. I won some expensive Java software at the end of an online seminar. Later that month, I won a microchip–enabled pinky ring (a memento from a 1998 Java conference). The following year I won a wireless PDA. Just last week, I won a fancy business-class printer.

I never spend money to enter any contests. All these winnings are freebies. When the national computer science educators' conference met in Reno, Nevada, my colleagues convinced me to try the slot machines. I lost $23, and then I won back $18. At that point, I stopped playing. I wanted to quit while I was only $5 behind.

That's why my writing a Java program about slot machines is such a strange occurrence. A typical slot machine has three reels, with each reel having about 20 symbols. But to illustrate this section's ideas, I don't need 20 symbols. Instead I use four symbols — a cherry, a lemon, a kumquat, and a rutabaga.

Creating an enhanced for loop

When you play my simplified slot machine, you can spin any one of over 60 combinations — cherry+cherry+kumquat, rutabaga+rutabaga+rutabaga, or whatever. This chapter's goal is to list all possible combinations. But first, I show you another kind of loop. Listing 15-5 defines an enum type for a slot machine's symbols and displays a list of the symbols. (For an introduction to enum types, see Chapter 10.)

Listing 15-5: Slot Machine Symbols

```
import static java.lang.System.out;

class ListSymbols {

  enum Symbol {
    cherry, lemon, kumquat, rutabaga
  }

  public static void main(String args[]) {
    for (Symbol leftReel : Symbol.values()) {
      out.println(leftReel);
    }
  }
}
```

Listing 15-5 uses Java's *enhanced for loop*. The word "enhanced" means "enhanced compared with the loops in earlier versions of Java." The enhanced `for` loop was introduced in Java version 5.0. If you run Java version 1.4.2 (or something like that), then you can't use an enhanced `for` loop.

Here's the format of the enhanced `for` loop:

```
for (TypeName variableName : RangeOfValues) {
    Statements
}
```

Here's how the loop in Listing 15-5 follows the format:

- **In Listing 15-5, the word `Symbol` is the name of a type.**

  The `int` type describes values like -1, 0, 1, and 2. The `boolean` type describes the values `true` and `false`. And (because of the code in Listing 15-5) the `Symbol` type describes the values `cherry`, `lemon`, `kumquat`, and `rutabaga`. For more information on enum types like `Symbol`, see Chapter 10.

- **In Listing 15-5, the word `leftReel` is the name of a variable.**

  The loop in Listing 15-1 defines `count` to be an `int` variable. Similarly, the loop in Listing 15-5 defines `leftReel` to be a `Symbol` variable. So in theory, the variable `leftReel` can take on any of the four `Symbol` values.

  By the way, I call this variable `leftReel` because the code lists all the symbols that can appear on the leftmost of the slot machine's three reels. Because all three of the slot machine's reels have the same symbols, I may also have named this variable `middleReel` or `rightReel`. But on second thought, I'll save the names `middleReel` and `rightReel` for a later example.

- **In Listing 15-5, the expression `Symbol.values()` stands for the four values in Listing 15-5.**

  To quote myself in the previous bullet, "in theory, the variable `leftReel` can take on any of the four `Symbol` values." Well, the *`RangeOfValues`* part of the `for` statement turns theory into practice. This third item inside the parentheses says, "Have as many loop iterations as there are `Symbol` values and have the `leftReel` variable take on a different `Symbol` value during each of the loop's iterations."

  So the loop in Listing 15-5 undergoes four iterations — an iteration in which `leftReel` has value `cherry`, another iteration in which `leftReel` has value `lemon`, a third iteration in which `leftReel` has value `kumquat`, and a fourth iteration in which `leftReel` has value `rutabaga`. During each iteration, the program prints the `leftReel` variable's value. The result is in Figure 15-8.

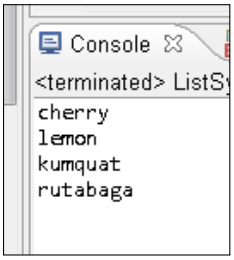


Figure 15-8: The output of the code in Listing 15-5.

In general, a *`someEnumTypeName`*`.values()` expression stands for the set of values that a particular enum type's variable can have. For example, in Listing 10-7 you may use the expression `WhoWins.values()` to refer to the `home`, `visitor`, and `neither` values.

The difference between a type's name (like `Symbol`) and the type's values (as in `Symbol.values()`) is really subtle. Fortunately, you don't have to worry about the difference. As a beginning programmer, you can just use the `.values()` suffix in an enhanced loop's *`RangeOfValues`* part.

Nesting the enhanced for loops

Listing 15-5 solves a simple problem in a very elegant way. So after reading about Listing 15-5, you ask about more complicated problems. "Can I list all possible three-reel combinations of the slot machine's four symbols?" Yes, you can. Listing 15-6 shows you how to do it.

Listing 15-6: Listing the Combinations

```
import static java.lang.System.out;

class ListCombinations {

    enum Symbol {
        cherry, lemon, kumquat, rutabaga
    }

    public static void main(String args[]) {

        for (Symbol leftReel : Symbol.values()) {
            for (Symbol middleReel : Symbol.values()) {
                for (Symbol rightReel : Symbol.values()) {
                    out.print(leftReel);
                    out.print(" ");
                    out.print(middleReel);
                    out.print(" ");
                    out.println(rightReel);
                }
            }
        }
    }
}
```

When you run the program in Listing 15-6, you get 64 lines of output. Some of those lines are shown in Figure 15-9.

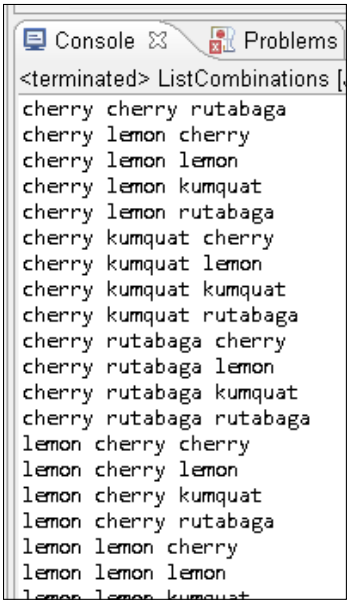


Figure 15-9: The first several lines of output from the code in Listing 15-6.

Like the code in Listing 15-3, the program in Listing 15-6 contains a loop within a loop. In fact, Listing 15-6 has a loop within a loop within a loop. Here's the strategy in Listing 15-6:

```
for (each of the 4 symbols that
     can appear on the left reel),

    for (each of the 4 symbols that
         can appear on the middle reel),

        for (each of the 4 symbols that
             can appear on the right reel),

            display the three reels' symbols.
```

So you start the outer loop with the `cherry` symbol. Then you march on to the middle loop and begin that loop with the `cherry` symbol. Then you proceed to the inner loop and pick the `cherry` (pun intended). At last, with each loop tuned to the `cherry` setting, you display the `cherry cherry cherry` combination (see Figure 15-10).

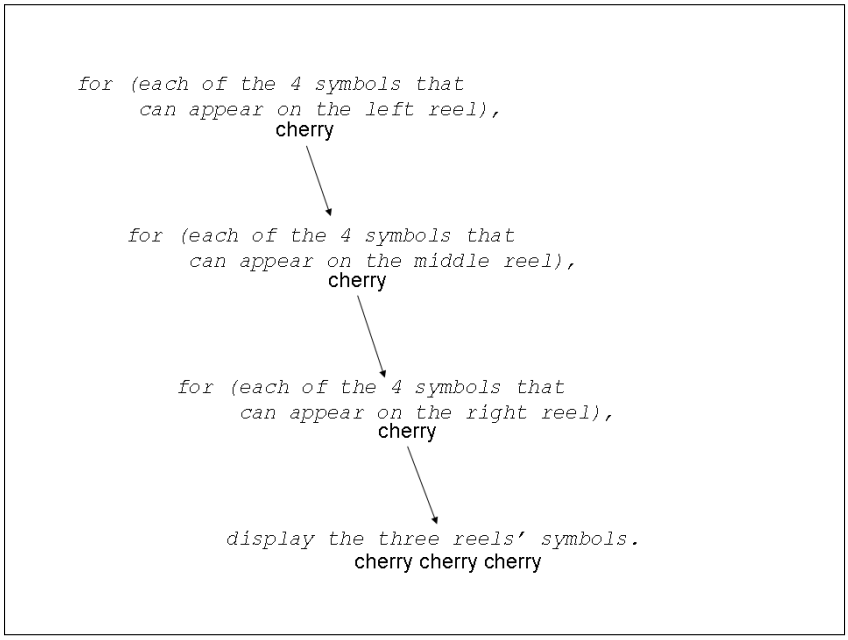

for (each of the 4 symbols that
can appear on the left reel),
cherry
for (each of the 4 symbols that
can appear on the middle reel),
cherry
for (each of the 4 symbols that
can appear on the right reel),
cherry
display the three reels' symbols.
cherry cherry cherry

Figure 15-10: Entering loops for the first time in the program of Listing 15-6.

After displaying `cherry cherry cherry`, you continue with other values of the innermost loop. That is, you change the right reel's value from `cherry` to `lemon` (see Figure 15-11). Now the three reels' values are `cherry cherry lemon`, so you display these values on the screen. (See the second line in Figure 15-9.)

After exhausting the four values of the innermost (right reel) loop, you jump out of that innermost loop. But the jump puts you back to the top of the middle loop, where you change the value of `middleReel` from `cherry` to `lemon`. Now the values of `leftReel` and `middleReel` are `cherry` and `lemon`, respectively (see Figure 15-12).

Having changed to `lemon` on the middle loop, you go barreling again into the innermost loop. As if you'd never seen this inner loop before, you set the loop's variable to `cherry` (see Figure 15-13).

After displaying the tasty `cherry lemon cherry` combination, you start changing the values of the innermost loop (see Figure 15-14).

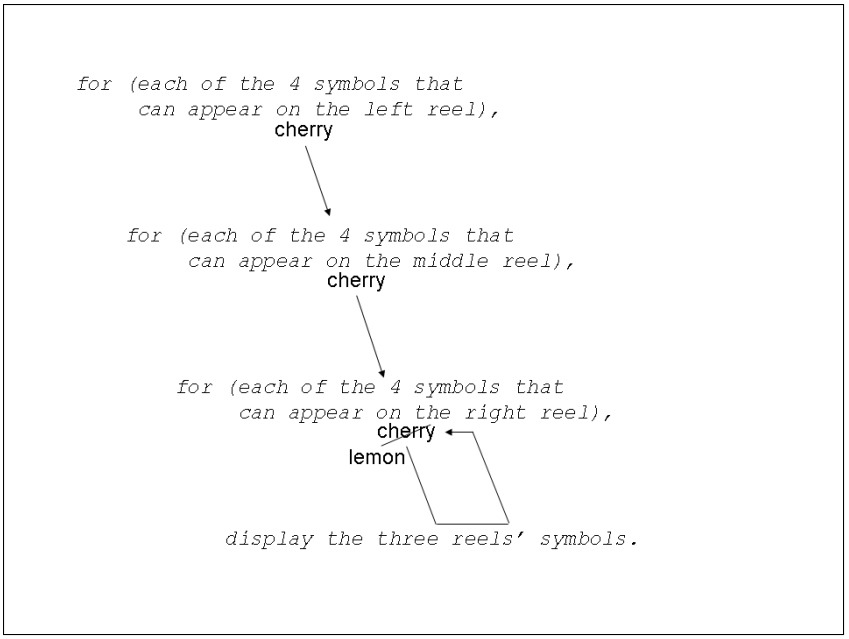

for (each of the 4 symbols that
can appear on the left reel),
cherry
for (each of the 4 symbols that
can appear on the middle reel),
cherry
for (each of the 4 symbols that
can appear on the right reel),
cherry
lemon
display the three reels' symbols.

Figure 15-11: Changing from cherry to lemon in the innermost loop.

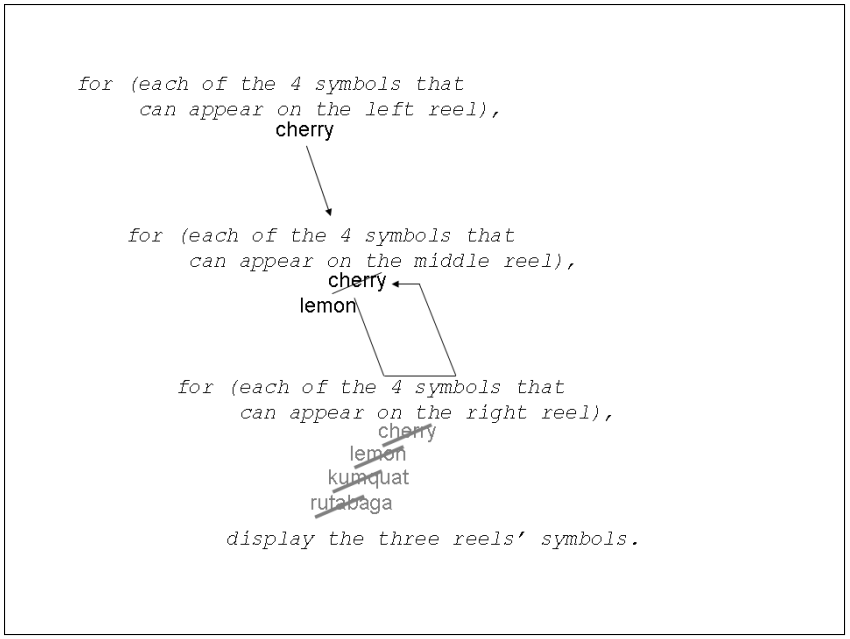


Figure 15-12: Changing from cherry to lemon in the middle loop.

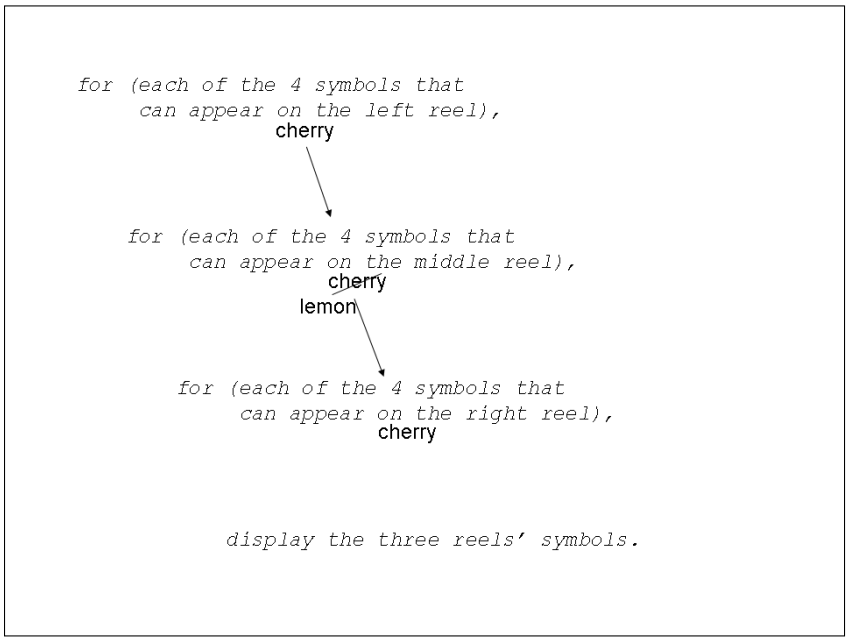

for (each of the 4 symbols that
can appear on the left reel),
cherry
for (each of the 4 symbols that
can appear on the middle reel),
cherry
lemon
for (each of the 4 symbols that
can appear on the right reel),
cherry
display the three reels' symbols.

Figure 15-13: Restarting the inner loop.

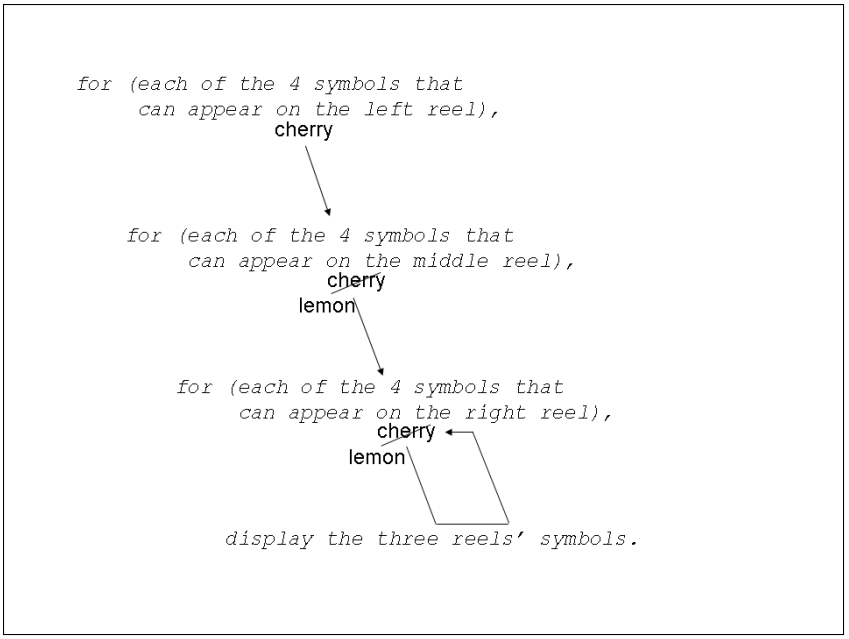
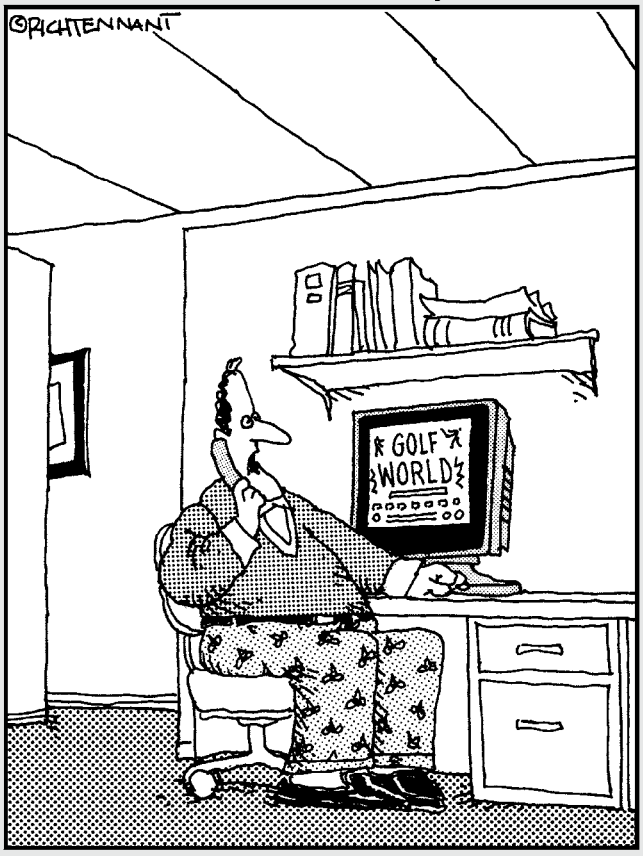

for (each of the 4 symbols that
can appear on the left reel),
cherry
for (each of the 4 symbols that
can appear on the middle reel),
cherry
lemon
for (each of the 4 symbols that
can appear on the right reel),
cherry
lemon
display the three reels' symbols.

Figure 15-14: Traveling a second time through the innermost loop.

The loop keeps going until it displays all 64 combinations. Whew!

Part IV
Using Program Units

The 5th Wave By Rich Tennant



"I can't really explain it, but every time I animate someone swinging a golf club, a little divot of code comes up missing on the home page."

In this part . . .

Way back in the Elvis Era, people thought that computer programs should be big lists of instructions. Then, during the Groovy Sixties, people decided to modularize their programs. A typical program consisted of several methods (like the `main` methods in this book's examples). Finally, during the Weighty Eighties, programmers grouped methods and other things into units called *objects*.

Far from being the flavor of the month, object-oriented programming has become the backbone of modern computing. This part of the book tells you all about it.

Chapter 16

Using Loops and Arrays

In This Chapter

- Using for loops to the max
- Storing many values in a single variable
- Working with groups of values

This chapter has seven illustrations. For these illustrations, the people at Wiley Publishing insist on following numbering: Figure 16-1, Figure 16-2, Figure 16-3, Figure 16-4, Figure 16-5, Figure 16-6, and Figure 16-7. But I like a different kind of numbering. I'd like to number the illustrations `figure[0]`, `figure[1]`, `figure[2]`, `figure[3]`, `figure[4]`, `figure[5]`, and `figure[6]`. In this chapter, you'll find out why.

Some Loops in Action

The Java Motel, with its ten comfortable rooms, sits in a quiet place off the main highway. Aside from a small, separate office, the motel is just one long row of ground floor rooms. Each room is easily accessible from the spacious front parking lot.

Oddly enough, the motel's rooms are numbered 0 through 9. I could say that the numbering is a fluke — something to do with the builder's original design plan. But the truth is, starting with 0 makes the examples in this chapter easier to write.

You, as the Java Motel's manager, store occupancy data in a file on your computer's hard drive. The file has one entry for each room in the motel. For example, in Figure 16-1, Room 0 has one guest, Room 1 has four guests, Room 2 is empty, and so on.

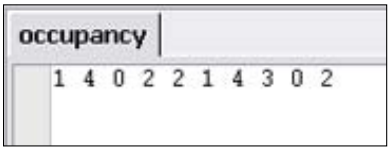


Figure 16-1: Occupancy data for the Java Motel.

You want a report showing the number of guests in each room. Because you know how many rooms you have, this problem begs for a `for` loop. The code to solve this problem is in Listing 16-1, and a run of the code is shown in Figure 16-2.

Listing 16-1: A Program to Generate an Occupancy Report

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
import static java.lang.System.out;

class ShowOccupancy {

    public static void main(String args[])
                          throws FileNotFoundException {

        Scanner diskScanner =
            new Scanner(new File("occupancy"));

        out.println("Room\tGuests");

        for (int roomNum = 0; roomNum < 10; roomNum++) {
            out.print(roomNum);
            out.print("\t");
            out.println(diskScanner.nextInt());
        }
    }
}
```

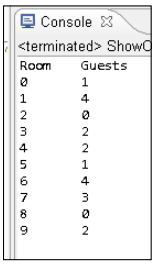


Figure 16-2: Running the code in Listing 16-1.

Listing 16-1 uses a `for` loop — a loop of the kind described in Chapter 15. As the `roomNum` variable's value marches from 0 to 9, the program displays one number after another from the `occupancy` file. To read more about getting numbers from a disk file like my `occupancy` file, see Chapter 13.

This example's input file is named `occupancy` — not `occupancy.txt`. If you use Windows Notepad to make an `occupancy` file, you must use quotation marks in the Save As dialog box's File Name field. That is, you must type `"occupancy"` (with quotation marks) in the File Name field. If you don't surround the name with quotation marks, then Notepad adds a default extension to the file's name (turning `occupancy` into `occupancy.txt`). A similar issue applies to the Macintosh's TextEdit program. By default, TextEdit adds the `.rtf` extension to each new file. To override the `.rtf` default for a particular file, select Format➪Make Plain Text. To override the default for all newly created files, choose TextEdit➪Preferences. Then, in the Format part of the Preferences dialog's New Document tab, select Plain Text.

Deciding on a loop's limit at runtime

On occasion, you may want a more succinct report than the one in Figure 16-2. "Don't give me a long list of rooms," you say. "Just give me the number of guests in Room 3." To get such a report, you need a slightly smarter program. The program is in Listing 16-2, with runs of the program shown in Figure 16-3.

Listing 16-2: Report on One Room Only, Please

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
import static java.lang.System.out;

public class ShowOneRoomOccupancy {

   public static void main(String args[])
                        throws FileNotFoundException {

      Scanner myScanner = new Scanner(System.in);
      Scanner diskScanner =
            new Scanner(new File("occupancy"));
      int whichRoom;

      out.print("Which room? ");
      whichRoom = myScanner.nextInt();
```

(continued)

Listing 16-2 ***(continued)***

```
        for (int roomNum = 0;
              roomNum < whichRoom; roomNum++){

           diskScanner.nextInt();
        }

        out.print("Room ");
        out.print(whichRoom);
        out.print(" has ");
        out.print(diskScanner.nextInt());
        out.println(" guest(s).");
    }
}
```

```
Which room? 3
Room 3 has 2 guest(s).


Which room? 5
Room 5 has 1 guest(s).


Which room? 8
Room 8 has 0 guest(s).


Which room? 10
Room 10 has Exception in thread "main" java.util.NoSuchElementException
        at java.util.Scanner.throwFor(Scanner.java:817)
        at java.util.Scanner.next(Scanner.java:1431)
        at java.util.Scanner.nextInt(Scanner.java:2040)
        at java.util.Scanner.nextInt(Scanner.java:2000)
        at ShowOneRoomOccupancy.main(ShowOneRoomOccupancy.java:26)
```

Figure 16-3: A few one-room reports.

If Listing 16-2 has a moral, it's that the number of `for` loop iterations can vary from one run to another. The loop in Listing 16-2 runs on and on as long as the counting variable `roomNum` is less than a room number specified by the user. When the `roomNum` is the same as the number specified by the user (that is, when `roomNum` is the same as `whichRoom`), the computer jumps out of the loop. Then the computer grabs one more `int` value from the `occupancy` file and displays that value on the screen.

As you stare at the runs in Figure 16-3, it's important to remember the unusual numbering of rooms. Room 3 has two guests because Room 3 is the *fourth* room in the `occupancy` file of Figure 16-1. That's because the motel's rooms are numbered 0 through 9.

Grabbing input here and there

Listing 16-2 illustrates some pithy issues surrounding the input of data. For one thing, the program gets input from both the keyboard and a disk file. (The program gets a room number from the keyboard. Then the program gets the number of guests in that room from the `occupancy` file.) To make this happen, Listing 16-2 sports two `Scanner` declarations — one to declare `myScanner`, and a second to declare `diskScanner`.

Later in the program, the call `myScanner.nextInt` reads from the keyboard, and `diskScanner.nextInt` reads from the file. Within the program, you can read from the keyboard or the disk as many times as you want. You can even intermingle the calls — reading once from the keyboard, then three times from the disk, then twice from the keyboard, and so on. All you have to do is remember to use `myScanner` whenever you read from the keyboard and use `diskScanner` whenever you read from the disk.

Another interesting tidbit in Listing 16-2 concerns the `occupancy` file. Many of this chapter's examples read from an `occupancy` file, and I use the same data in each of the examples. (I use the data shown in Figure 16-1.) To run an example, I copy the occupancy file from one Eclipse project to another. (Before running the code in Listing 16-2, I go to my old `16-01` project in Eclipse's Package Explorer. I right-click the `occupancy` file in the `16-01` project and select Copy from the context menu. Then I right-click the new `16-02` project branch and select Paste from the context menu. As usual, Mac users do control-click instead of right-click.)

In real life, having several copies of a data file can be dangerous. You can modify one copy and then accidentally read out-of-date data from a different copy. Sure, you should have backup copies, but you should have only one "master" copy — the copy from which all programs get the same input.

So in a real-life program, you don't copy the occupancy file from one project to another. What do you do instead? You put an `occupancy` file in one place on your hard drive and then have each program refer to the file using the names of the file's directories. For example, if your `occupancy` file is in the `c:\Oct\22` directory, you write

```
Scanner diskScanner =
 new Scanner(new File(
 "c:\\oct\\22\\occupancy"));
```

A sidebar in Chapter 13 has more details about filenames and double backslashes.

Using all kinds of conditions in a for loop

Look at the run in Figure 16-3 and notice the program's awful behavior when the user mistakenly asks about a nonexistent room. The motel has no Room 10. If you ask for the number of guests in Room 10, the program tries to read more numbers than the `occupancy` file contains. This unfortunate attempt causes a `NoSuchElementException`.

Listing 16-3 fixes the end-of-file problem.

Listing 16-3: A More Refined Version of the One-Room Code

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
import static java.lang.System.out;

public class BetterShowOneRoom {

   public static void main(String args[])
                        throws FileNotFoundException {

      Scanner myScanner = new Scanner(System.in);
      Scanner diskScanner =
            new Scanner(new File("occupancy"));
      int whichRoom;

      out.print("Which room? ");
      whichRoom = myScanner.nextInt();

      for (int roomNum = 0;
            roomNum < whichRoom && diskScanner.hasNext();
            roomNum++) {

         diskScanner.nextInt();
      }

      if (diskScanner.hasNext()) {
         out.print("Room ");
         out.print(whichRoom);
         out.print(" has ");
         out.print(diskScanner.nextInt());
         out.println(" guest(s).");
      }
   }
}
```

The code in Listing 16-3 isn't earth shattering. To get this code, you take the code in Listing 16-2 and add a few tests for the end of the `occupancy` file. You perform the `diskScanner.hasNext` test before each call to `nextInt`. That way, if the call to `nextInt` is doomed to failure, you catch the potential failure before it happens. A few test runs of the code in Listing 16-3 are shown in Figure 16-4.

In Listing 16-3, I want to know if the occupancy file contains any more data (any data that I haven't read yet). So I call the `Scanner` class's `hasNext` method. The `hasNext` method looks ahead to see whether I can read any kind of data — an `int` value, a `double` value, a word, a `boolean`, or whatever. That's okay for this section's example, but in some situations, you need to be

pickier about your input data. For example, you may want to know if you can call `nextInt` (as opposed to `nextDouble` or `nextLine`). Fortunately, Java has methods for your pickiest input needs. The code `if (diskScanner.hasNextInt())` tests to see whether you can read an `int` value from the disk file. Java also has methods like `hasNextLine`, `hasNextDouble`, and so on. For more information on the plain old `hasNext` method, see Chapter 14.

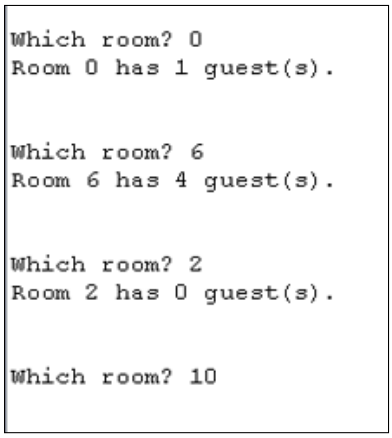


Figure 16-4: The bad room number 10 gets no response.

Listing 16-3 has a big fat condition to keep the `for` loop going:

```
for (int roomNum = 0;
      roomNum < whichRoom && diskScanner.hasNext();
      roomNum++) {
```

Many `for` loop conditions are simple "less-than" tests, but there's no rule saying that all `for` loop conditions have to be so simple. In fact, any expression can be a `for` loop's condition, as long as the expression has value `true` or `false`. The condition in Listing 16-3 combines a "less than" with a call to the `Scanner` class's `hasNext` method.

Reader, Meet Arrays; Arrays, Meet the Reader

A weary traveler steps up to the Java Motel's front desk. "I'd like a room," says the traveler. So the desk clerk runs a report like the one in Figure 16-2. Noticing the first vacant room in the list, the clerk suggests Room 2. "I'll take it," says the traveler.

It's so hard to get good help these days. How many times have you told the clerk to fill the higher numbered rooms first? The lower numbered rooms are older, and they are badly in need of repair. For example, Room 3 has an indoor pool. (The pipes leak, so the carpet is soaking wet.) Room 2 has no heat (not in wintertime, anyway). Room 1 has serious electrical problems (so, for that room, you always get payment in advance). Besides, Room 8 is vacant, and you charge more for the higher numbered rooms.

Here's where a subtle change in presentation can make a big difference. You need a program that lists vacant rooms in reverse order. That way, Room 8 catches the clerk's eye before Room 2 does.

Think about strategies for a program that displays data in reverse. With the input from Figure 16-1, the program's output should look like the display shown in Figure 16-5.

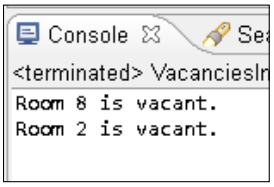


Figure 16-5: A list of vacant rooms, with higher numbered rooms shown first.

Here's the first (bad) idea for a programming strategy:

```
Get the last value in the occupancy file.
If the value is 0, print the room number.

Get the next-to-last value in the occupancy file.
If the value is 0, print the room number.

...And so on.
```

With some fancy input/output programs, this strategy may be workable. But no matter what input/output program you use, jumping directly to the end or to the middle of a file is a big pain in the boot. It's especially bad if you plan to jump repeatedly. So go back to the drawing board and think of something better.

Here's an idea! Read all the values in the `occupancy` file and store each value in a variable of its own. Then you step through the variables in reverse order, displaying a room number when it's appropriate to do so.

This idea works, but the code is so ugly that I refuse to dignify it by calling it a "Listing." No, this is just a "see the following code" kind of thing. So please, see the following ugly code:

```
/*
 * Ugh! I can't stand this ugly code!
 */
guestsIn0 = diskScanner.nextInt();
guestsIn1 = diskScanner.nextInt();
guestsIn2 = diskScanner.nextInt();
guestsIn3 = diskScanner.nextInt();
guestsIn4 = diskScanner.nextInt();
guestsIn5 = diskScanner.nextInt();
guestsIn6 = diskScanner.nextInt();
guestsIn7 = diskScanner.nextInt();
guestsIn8 = diskScanner.nextInt();
guestsIn9 = diskScanner.nextInt();

if (guestsIn9 == 0) {
    System.out.println(9);
}
if (guestsIn8 == 0) {
    System.out.println(8);
}
if (guestsIn7 == 0) {
    System.out.println(7);
}
if (guestsIn6 == 0) {

// ... And so on.
```

What you're lacking is a uniform way of naming ten variables. That is, it would be nice to write

```
/*
 * Nice idea, but this is not real Java code:
 */

//Read forwards
for (int roomNum = 0; roomNum < 10; roomNum++) {
    guestsIn roomNum = diskScanner.nextInt();
}

//Write backwards
for (int roomNum = 9; roomNum >= 0; roomNum--) {
    if (guestsIn roomNum == 0) {
        System.out.println(roomNum);
    }
}
```

Well, you can write loops of this kind. All you need are some square brackets. When you add square brackets to the idea shown in the preceding code, you get what's called an array. An *array* is a row of values, like the row of rooms in a one-floor motel. To picture the array, just picture the Java Motel:

- First, picture the rooms, lined up next to one another.
- Next, picture the same rooms with their front walls missing. Inside each room, you can see a certain number of guests.
- If you can, forget that the two guests in Room 9 are putting piles of bills into a big briefcase. Ignore the fact that the guest in Room 5 hasn't moved away from the TV set in a day and a half. Instead of all these details, just see numbers. In each room, see a number representing the count of guests in that room. (If freeform visualization isn't your strong point, then take a look at Figure 16-6.)

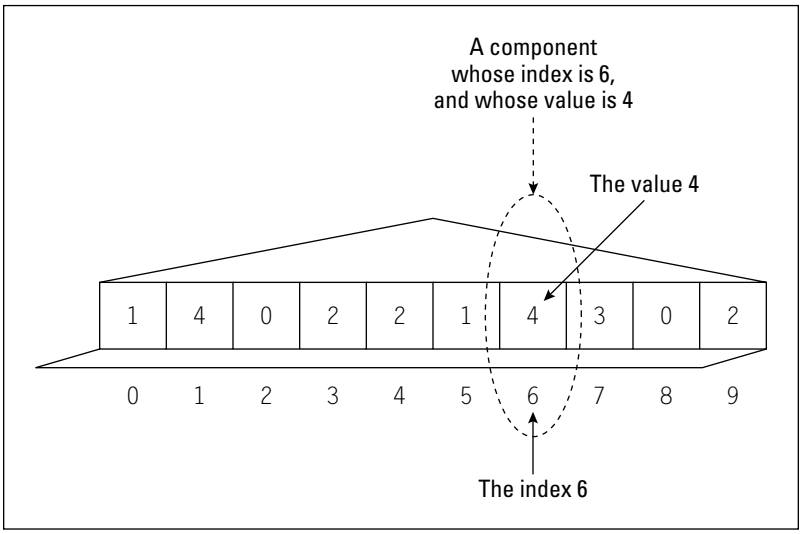


Figure 16-6: An abstract snapshot of rooms in the Java Motel.

In the lingo of Java programming, the entire row of rooms is called an *array*. Each room in the array is called a *component* of the array (also known as an array *element*). Each component has two numbers associated with it:

- **Index:** In the case of the Java Motel array, the index is the room number (a number from 0 to 9).
- **Value:** In the Java Motel array, the value is the number of guests in a given room (a number stored in a component of the array).

Using an array saves you from having to declare ten separate variables: `guestsIn0`, `guestsIn1`, `guestsIn2`, and so on. To declare an array with ten values in it, you can write two fairly short lines of code:

```
int guestsIn[];
guestsIn = new int[10];
```

You can even squish these two lines into one longer line:

```
int guestsIn[] = new int[10];
```

In either of these code snippets, notice the use of the number `10`. This number tells the computer to make the `guestsIn` array have ten components. Each component of the array has a name of its own. The starting component is named `guestsIn[0]`, the next is named `guestsIn[1]`, and so on. The last of the ten components is named `guestsIn[9]`.

In creating an array, you always specify the number of components. The array's indices always start with 0 and end with the number that's one fewer than the total number of components. For example, if your array has ten components (and you declare the array with `new int[10]`), then the array's indices go from 0 to 9.

Storing values in an array

After you've created an array, you can put values into the array's components. For example, the guests in Room 6 are fed up with all those mint candies that you put on peoples' beds. So they check out, and Room 6 becomes vacant. You should put the value 0 into the 6 component. You can do it with this assignment statement:

```
guestsIn[6] = 0;
```

On one weekday, business is awful. No one's staying at the motel. But then you get a lucky break. A big bus pulls up to the motel. The side of the bus has a sign that says "Loners' Convention." Out of the bus come 25 people, each walking to the motel's small office, none paying attention to the others who were on the bus. Each person wants a private room. Only 10 of them can stay at the Java Motel, but that's okay, because you can send the other 15 loners down the road to the old C-Side Resort and Motor Lodge.

Anyway, to register ten of the loners into the Java Motel, you put one guest in each of your ten rooms. Having created an array, you can take advantage of the array's indexing and write a `for` loop, like this:

```
for (int roomNum = 0; roomNum < 10; roomNum++) {
    guestsIn[roomNum] = 1;
}
```

This loop takes the place of ten assignment statements because the computer executes the statement `guestsIn[roomNum] = 1` ten times. The first time around, the value of `roomNum` is 0, so in effect, the computer executes

```
guestsIn[0] = 1;
```

In the next loop iteration, the value of `roomNum` is 1, so the computer executes the equivalent of the following statement:

```
guestsIn[1] = 1;
```

During the next iteration, the computer behaves as if it's executing

```
guestsIn[2] = 1;
```

And so on. When `roomNum` gets to be 9, the computer executes the equivalent of the following statement:

```
guestsIn[9] = 1;
```

Notice that the loop's counter goes from 0 to 9. Compare this with Figure 16-6 and remember that the indices of an array go from 0 to one fewer than the number of components in the array. Looping with room numbers from 0 to 9 covers all the rooms in the Java Motel.

When you work with an array, and you step through the array's components using a `for` loop, you normally start the loop's counter variable at 0. To form the condition that tests for another iteration, you often write an expression like `roomNum < arraySize`, where *`arraySize`* is the number of components in the array.

Creating a report

The code to create the report in Figure 16-5 is shown in Listing 16-4. This new program uses the idea in the world's ugliest code (the code from several pages back with variables `guestsIn0`, `guestsIn1`, and so on). But instead of having ten separate variables, Listing 16-4 uses an array.

Listing 16-4: Traveling through Data Both Forward and Backward

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

class VacanciesInReverse {

    public static void main(String args[])
        throws FileNotFoundException {

        Scanner diskScanner =
            new Scanner(new File("occupancy"));
        int guestsIn[];
        guestsIn = new int[10];

        for (int roomNum = 0; roomNum < 10; roomNum++) {
            guestsIn[roomNum] = diskScanner.nextInt();
        }

        for (int roomNum = 9; roomNum >= 0; roomNum--) {
            if (guestsIn[roomNum] == 0) {
                System.out.print("Room ");
                System.out.print(roomNum);
                System.out.println(" is vacant.");
            }
        }
    }
}
```

Notice the stuff in parentheses in the `VacanciesInReverse` program's second `for` loop. It's easy to get these things wrong. You're aiming for a loop that checks Room 9, then Room 8, and so on.

```
if (guestsIn[9] == 0) {
    System.out.print(roomNum);
}
if (guestsIn[8] == 0) {
    System.out.print(roomNum);
}
if (guestsIn[7] == 0) {
    System.out.print(roomNum);
}

...And so on, until you get to...

if (guestsIn[0] == 0) {
    System.out.print(roomNum);
}
```

Some observations about the code:

- The loop's counter must start at 9:

```
for (int roomNum = 9; roomNum >= 0; roomNum--)
```

- Each time through the loop, the counter goes *down* by one:

```
for (int roomNum = 9; roomNum >= 0; roomNum--)
```

- The loop keeps going as long as the counter is *greater than or equal to* 0:

```
for (int roomNum = 9; roomNum >= 0; roomNum--)
```

Think through each of these three items, and you'll write a perfect `for` loop.

Working with Arrays

Earlier in this chapter, a busload of loners showed up at your motel. When they finally left, you were glad to get rid them, even if it meant having all your rooms empty for a while. But now, another bus pulls into the parking lot. This bus has a sign that says "Gregarian Club." Out of the bus come 50 people, each more gregarious than the next. Now everybody in your parking lot is clamoring to meet everyone else. While they meet and greet, they're all frolicking toward the front desk, singing the club's theme song. (Oh no! It's the Gregarian Chant!)

The first five Gregarians all want Room 7. It's a tight squeeze, but you were never big on fire codes, anyway. Next comes a group of three with a yen for Room 0. (They're computer programmers, and they think the room number is cute.) Then there's a pack of four Gregarians who want Room 3. (The in-room pool sounds attractive to them.)

With all this traffic, you better switch on your computer. You start a program that enables you to enter new occupancy data. The program has five parts:

- **Create an array and then put 0 in each of the array's components.**

  When the Loners' Club members left, the motel was suddenly empty. (Heck, even before the Loners' Club members left, the motel seemed empty.) To declare an array and fill the array with zeros, you execute code of the following kind:

```
int guestsIn[];
guestsIn = new int[10];

for (int roomNum = 0; roomNum < 10; roomNum++) {
    guestsIn[roomNum] = 0;
}
```

- **Get a room number and then get the number of guests who will be staying in that room.**

  Reading numbers typed by the user is pretty humdrum stuff. Do a little prompting and a little `nextInt` calling, and you're all set:

```
out.print("Room number: ");
whichRoom = myScanner.nextInt();
out.print("How many guests? ");
numGuests = myScanner.nextInt();
```

- **Use the room number and the number of guests to change a value in the array.**

  Earlier in this chapter, to put one guest in Room 2, you executed

```
guestsIn[2] = 1;
```

  So now, you have two variables — `numGuests` and `whichRoom`. Maybe `numGuests` is 5, and `whichRoom` is 7. To put `numGuests` in `whichRoom` (that is, to put 5 guests in Room 7), you can execute

```
guestsIn[whichRoom] = numGuests;
```

  That's the crucial step in the design of your new program.

- **Ask the user if the program should keep going.**

  Are there more guests to put in rooms? To find out, execute this code:

```
    out.print("Do another? ");
} while (myScanner.
         findWithinHorizon(".",0).charAt(0) == 'Y');
```

- **Display the number of guests in each room.**

  No problem! You already did this. You can steal the code (almost verbatim) from Listing 16-1:

```
out.println("Room\tGuests");
for (int roomNum = 0; roomNum < 10; roomNum++) {
    out.print(roomNum);
    out.print("\t");
    out.println(guestsIn[roomNum]);
}
```

  The only difference between this latest code snippet and the stuff in Listing 16-1 is that this new code uses the `guestsIn` array. The first time through this loop, the code does

```
out.println(guestsIn[0]);
```

  displaying the number of guests in Room 0. The next time through the loop, the code does

```
out.println(guestsIn[1]);
```

displaying the number of guests in Room 1. The last time through the loop, the code does

```
out.println(guestsIn[9]);
```

That's perfect.

The complete program (with these five pieces put together) is in Listing 16-5. A run of the program is shown in Figure 16-7.

Listing 16-5: Storing Occupancy Data in an Array

```
import java.util.Scanner;
import static java.lang.System.out;

class AddGuests {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        int whichRoom, numGuests;
        int guestsIn[];
        guestsIn = new int[10];

        for (int roomNum = 0; roomNum < 10; roomNum++) {
            guestsIn[roomNum] = 0;
        }

        do {
            out.print("Room number: ");
            whichRoom = myScanner.nextInt();
            out.print("How many guests? ");
            numGuests = myScanner.nextInt();
            guestsIn[whichRoom] = numGuests;

            out.println();
            out.print("Do another? ");
        } while (myScanner.
              findWithinHorizon(".",0).charAt(0) == 'Y');

        out.println();
        out.println("Room\tGuests");
        for (int roomNum = 0; roomNum < 10; roomNum++) {
            out.print(roomNum);
            out.print("\t");
            out.println(guestsIn[roomNum]);
        }
    }
}
```

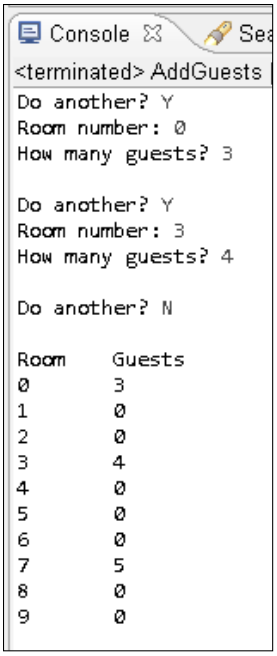


Figure 16-7: Running the code in Listing 16-5.

Hey! The program in Listing 16-5 is pretty big! It may be the biggest program so far in this book. But *big* doesn't necessarily mean *difficult.* If each piece of the program makes sense, you can create each piece on its own, and then put all the pieces together. Voilà! The code is manageable.

Looping in Style

Listing 15-6 uses an enhanced `for` loop to step through a bunch of values. In that program, the values belong to an enum type. Well, this chapter also deals with a bunch of values — namely, the values in an array. So you're probably not surprised if I show you an enhanced `for` loop that steps through an array's values.

To see such a loop, start with the code in Listing 16-5. The last loop in that program looks something like this:

```
for (int roomNum = 0; roomNum < 10; roomNum++) {
    out.println(guestsIn[roomNum]);
}
```

To turn this into an enhanced `for` loop, you make up a new variable name. (What about the name `howMany`? I like that name.) Whatever name you choose, the new variable ranges over the values in the `guestsIn` array.

```
for (int howMany : guestsIn) {
    out.println(howMany);
}
```

This enhanced loop uses the same format as the loop in Chapter 15.

```
for (TypeName variableName : RangeOfValues) {
    Statements
}
```

In Chapter 15, the *`RangeOfValues`* belongs to an enum type. But in this sidebar's example, the *`RangeOfValues`* belongs to an array.

Enhanced `for` loops are nice and concise. But don't be too anxious to use enhanced loops with arrays. This feature has some nasty limitations. For example, my new `howMany` loop doesn't display room numbers. I avoid room numbers because the room numbers in my `guestsIn` array are the indices 0 through 9. Unfortunately, an enhanced loop doesn't provide easy access to an array's indices.

And here's another unpleasant surprise. Start with the following loop from Listing 16-4:

```
for (int roomNum = 0; roomNum < 10; roomNum++) {
    guestsIn[roomNum] = diskScanner.nextInt();
}
```

Turn this traditional `for` loop into an enhanced `for` loop, and you get the following misleading code:

```
for (int howMany : guestsIn) {
    howMany = diskScanner.nextInt();  //Don't do this
}
```

The new enhanced loop doesn't do what you want it to do. This loop reads values from an input file and then dumps these values into the garbage can. In the end, the array's values remain unchanged.

It's sad but true. To make full use of an array, you have to fall back on Java's plain old `for` loop.

Chapter 17

Programming with Objects and Classes

In This Chapter

- Programming with class (and with style and finesse)
- Making objects from classes
- Joining the exclusive "I understand classes and objects" society

Chapters 6, 7, and 8 introduce Java's primitive types — things like `int`, `double`, `char`, and `boolean`. That's great, but how often does a real-world problem deal exclusively with such simple values? Consider an exchange between a merchant and a customer. The customer makes a purchase, which can involve item names, model numbers, credit card info, sales tax rates, and lots of other stuff.

In older computer programming languages, you treat an entire purchase like a big pile of unbundled laundry. Imagine a mound of socks, shirts, and other pieces of clothing. You have no basket, so you grab as much as you can handle. As you walk to the washer, you drop a few things — a sock here and a washcloth there. This is like the older way of storing the values in a purchase. In older languages, there's no purchase. There are only `double` values, `char` values, and other loose items. You put the purchase amount in one variable, the customer's name in another, and the sales tax data somewhere else. But that's awful. You tend to drop things on your way to the compiler. With small errors in a program, you can easily drop an amount here and a customer's name there.

So with laundry and computer programming, you're better off if you have a basket. The newer programming languages, like Java, allow you to combine values and make new, more useful kinds of values. For example, in Java, you can combine `double` values, `boolean` values, and other kinds of values to create something that you call a `Purchase`. Because your purchase info is all in one big bundle, keeping track of the purchase's pieces is easier. That's the start of an important computer programming concept — the notion of *object-oriented programming*.

Creating a Class

I start with a "traditional" example. The program in Listing 17-1 processes simple purchase data. Two runs of the program are shown in Figure 17-1.

Listing 17-1: Doing It the Old Fashioned Way

```
import java.util.Scanner;

class ProcessData {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        double amount;
        boolean taxable;
        double total;

        System.out.print("Amount: ");
        amount = myScanner.nextDouble();
        System.out.print("Taxable? (true/false) ");
        taxable = myScanner.nextBoolean();

        if (taxable) {
           total = amount * 1.05;
        } else {
            total = amount;
        }

        System.out.print("Total: ");
        System.out.println(total);
    }
}
```

```
Amount: 20.00
Taxable? (true/false) false
Total: 20.0

Amount: 20.00
Taxable? (true/false) true
Total: 21.0
```

Figure 17-1: Processing a customer's purchase.

If the output in Figure 17-1 looks funny, it's because I do nothing in the code to control the number of digits beyond the decimal point. So in the output, the value $20.00 looks like `20.0`. That's okay. I show you how to fix the problem in Chapter 18.

Reference types and Java classes

The code in Listing 17-1 involves a few simple values — `amount`, `taxable`, and `total`. So here's the main point of this chapter: By combining several simple values, you can get a single, more useful value. That's the way it works. You take some of Java's primitive types, whip them together to make a primitive type stew, and what do you get? You get a more useful type called a *reference type*. Listing 17-2 has an example.

Listing 17-2: What It Means to Be a Purchase

```
class Purchase {
    double amount;
    boolean taxable;
    double total;
}
```

The code in Listing 17-2 has no `main` method, so Eclipse can compile the code, but you can't run it. When you choose Run➪Run As in Eclipse's main menu, the resulting context menu has no Java Application entry. You can click the tiny Run As button in Eclipse's toolbar and then select Java Application. But then you get the message box shown in Figure 17-2. Because Listing 17-2 has no `main` method, there's no place to start the executing. (In fact, the code in Listing 17-2 has no statements at all. There's nothing to execute.)

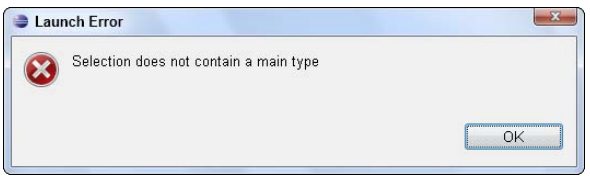


Figure 17-2: The code in Listing 17-2 has no main method.

Using a newly defined class

To do something useful with the code in Listing 17-2, you need a `main` method. You can put the `main` method in a separate file. Listing 17-3 shows you such a file.

Listing 17-3: Making Use of Your Purchase Class

```
import java.util.Scanner;

class ProcessPurchase {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        Purchase onePurchase = new Purchase();

        System.out.print("Amount: ");
        onePurchase.amount = myScanner.nextDouble();
        System.out.print("Taxable? (true/false) ");
        onePurchase.taxable = myScanner.nextBoolean();

        if (onePurchase.taxable) {
            onePurchase.total = onePurchase.amount * 1.05;
        } else {
            onePurchase.total = onePurchase.amount;
        }

        System.out.print("Total: ");
        System.out.println(onePurchase.total);
    }
}
```

The best way to understand the code in Listing 17-3 is to compare it, line by line, with the code in Listing 17-1. In fact, there's a mechanical formula for turning the code in Listing 17-1 into the code in Listing 17-3. Table 17-1 describes the formula.

Table 17-1 Converting Your Code to Use a Class

In Listing 17-1	*In Listing 17-3*

`double amount;`
`boolean taxable;`
`double total;`
	`Purchase onePurchase = new Purchase();`

`amount`	`onePurchase.amount`
`taxable`	`onePurchase.taxable`
`total`	`onePurchase.total`

The two programs (in Listings 17-1 and 17-3) do essentially the same thing, but one uses primitive variables, and the other leans on the `Purchase` code from Listing 17-2. Both programs have runs like the ones shown back in Figure 17-1.

Running code that straddles two separate files

From Eclipse's point of view, a project that contains two Java source files is no big deal. You create two classes in the same project, and then you choose Run➪Run As➪Java Application. Everything works the way you expect it to work.

The only time things become tricky is when you have two `main` methods in the one project. This section's example (Listings 17-2 and 17-3) doesn't suffer from that malady. But as you experiment with your code, you can easily add classes with additional main methods. You may also create a large application with several starting points.

When a project has more than one `main` method, Eclipse may prompt you and ask which class's `main` method you want to run. But sometimes Eclipse doesn't prompt you. Instead, Eclipse arbitrarily picks one of the `main` methods and ignores all the others. This can be very confusing. You add a `println` call to the wrong `main` method, and nothing appears in the Console view. Hey, what gives?

You can fix the problem by following these steps:

1. **Expand the project's branch in the Package Explorer.**
2. **Expand the src folder within the project's branch.**
3. **Expand the (default package) branch within the src branch.**

   The (default package) branch contains `.java` files.
4. **(In Windows:) Right-click the `.java` file whose `main` method you want to run. (On a Mac:) Control-click the `.java` file whose `main` method you want to run.**
5. **In the resulting context menu, choose Run As➪Java Application.**

You cannot run a project that has no `main` method. If you try, you get a message box like the one shown earlier in Figure 17-2.

Why bother?

On the surface, the code in Listing 17-3 is longer, more complicated, and harder to read. But think about a big pile of laundry. It may take time to find a basket and to shovel socks into the basket. But when you have clothes in the

basket, the clothes are much easier to carry. It's the same way with the code in Listing 17-3. When you have your data in a `Purchase` basket, it's much easier to do complicated things with purchases.

From Classes Come Objects

The code in Listing 17-2 defines a class. A *class* is a design plan; it describes the way in which you intend to combine and use pieces of data. For example, the code in Listing 17-2 announces your intention to combine `double`, `boolean`, and `double` values to make new `Purchase` values.

Classes are central to all Java programming. But Java is called an object-oriented language. Java isn't called a class-oriented language. In fact, no one uses the term class-oriented language. Why not?

Well, you can't put your arms around a class. A class isn't real. A class without an object is like a day without chocolate. If you're sitting in a room right now, glance at all the chairs in the room. How many chairs are in the room? Two? Five? Twenty? In a room with five chairs, you have five chair objects. Each chair (each object) is something real, something you can use, something you can sit on.

A language like Java has classes and objects. So what's the difference between a class and an object?

- An object is a thing.
- A class is a design plan for things of that kind.

For example, how would you describe what a chair is? Well, a chair has a seat, a back, and legs. In Java, you may write the stuff in Listing 17-4.

Listing 17-4: What It Means to Be a Chair

```
/*
 * This is real Java code, but this code
 *  cannot be compiled on its own:
 */

class Chair {
    FlatHorizonalPanel seat;
    FlatVerticalPanel back;
    LongSkinnyVerticalRods legs;
}
```

The preceding code is a design plan for chairs. The code tells you that each chair has three things. The code names the things (`seat`, `back`, and `legs`) and tells you a little bit about each thing. (For example, a seat is a `FlatHorizontalPanel`.) In the same way, the code in Listing 17-2 tells you that each purchase has three things. The code names the things (`amount`, `taxable`, and `total`) and tells you the primitive type of each thing.

So imagine some grand factory at the edge of the universe. While you sleep each night, this factory stamps out tangible objects — objects that you'll encounter during the next waking day. Tomorrow you'll go for an interview at the Sloshy Shoes Company. So tonight, the factory builds chairs for the company's offices. The factory builds chair objects, as shown in Figure 17-3, from the almost-real code in Listing 17-4.

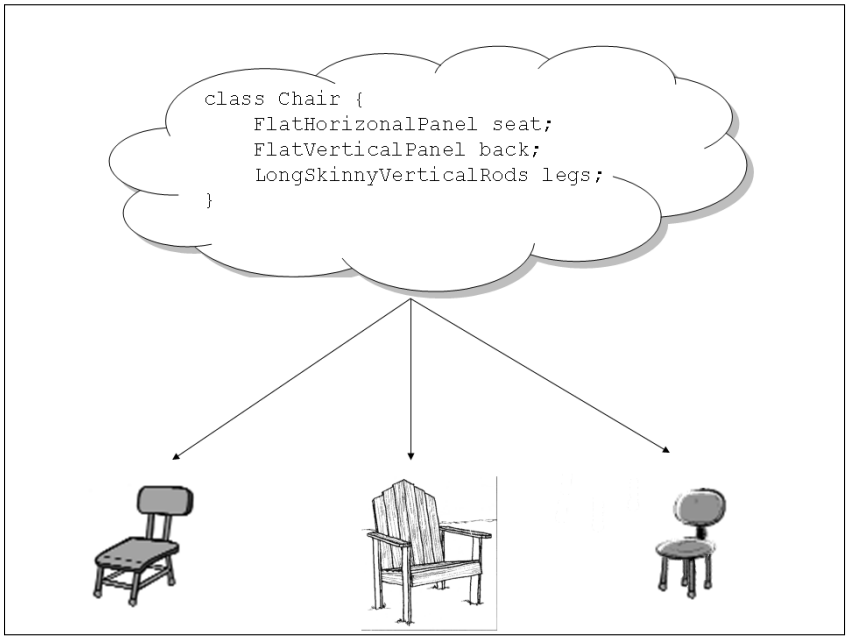


Figure 17-3: Chair objects from the Chair class.

In Listing 17-3, the line

```
Purchase onePurchase = new Purchase();
```

behaves like that grand factory at the edge of the universe. Instead of creating chair objects, that line in Listing 17-3 creates a purchase object, as shown in Figure 17-4. That Listing 17-3 line is a declaration with an initialization. Just as the line

```
int count = 0;
```

declares the `count` variable and sets `count` to 0, the line in Listing 17-3 declares the `onePurchase` variable and makes `onePurchase` point to a brand-new object. That new object contains three parts: an `amount` part, a `taxable` part, and a `total` part.

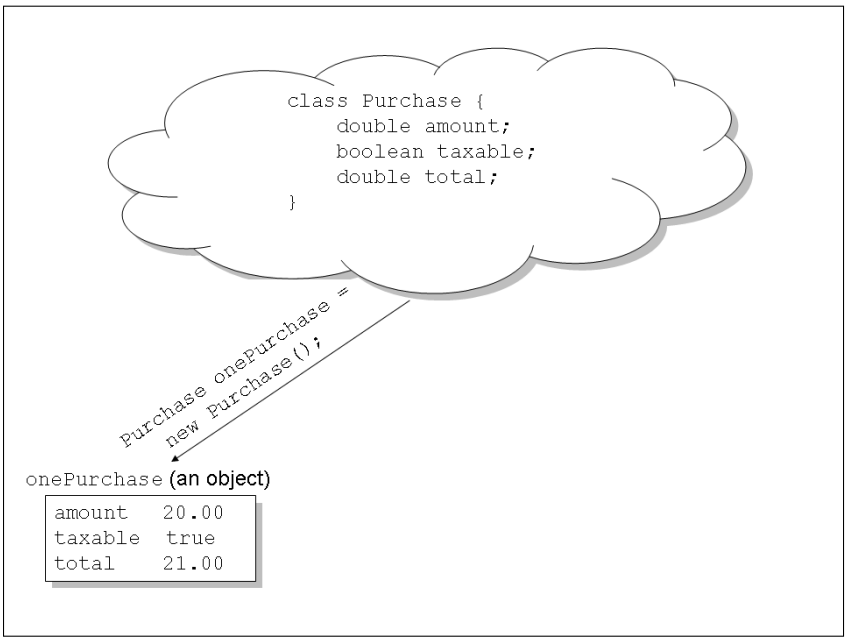


Figure 17-4: An object created from the Purchase class.

If you want to be picky, there's a difference between the stuff in Figure 17-4 and the action of the big bold statement in Listing 17-3. Figure 17-4 shows an object with the values `20.00`, `true`, and `21.00` stored in it. The statement in Listing 17-3 creates a new object, but it doesn't fill the object with useful values. Getting values comes later in Listing 17-3.

Understanding (or ignoring) the subtleties

Sometimes, when you refer to a particular object, you want to emphasize which class the object came from. Well, subtle differences in emphasis call

for big differences in terminology. So here's how Java programmers use the terminology:

- The bold line in Listing 17-3 creates a new *object*.
- The bold line in Listing 17-3 creates a new *instance of the* `Purchase` *class*.

The words *object* and *instance* are almost synonymous, but Java programmers never say "object of the `Purchase` class" (or if they do, they feel funny).

By the way, if you mess up this terminology and say something like "object of the `Purchase` class," then no one jumps down your throat. Everyone understands what you mean, and life goes on as usual. In fact, I often use a phrase like "`Purchase` object" to describe an instance of the `Purchase` class. The difference between object and instance isn't terribly important. But it's very important to remember that the words *object* and *instance* have the same meaning. (Okay! They have *nearly* the same meaning.)

Making reference to an object's parts

After you've created an object, you use dots to refer to the object's parts. For example, in Listing 17-3, I put a value into the `onePurchase` object's `amount` part with the following code:

```
onePurchase.amount = myScanner.nextDouble();
```

Later in Listing 17-3, I get the `amount` part's value with the following code:

```
onePurchase.total = onePurchase.amount * 1.05;
```

This dot business may look cumbersome, but it really helps programmers when they're trying to organize the code. In Listing 17-1, each variable is a separate entity. But in Listing 17-3, each use of the word `amount` is inextricably linked to the notion of a purchase. That's good.

Creating several objects

After you've created a `Purchase` class, you can create as many purchase objects as you want. For example, in Listing 17-5, I create three purchase objects.

Listing 17-5: Processing Purchases

```
import java.util.Scanner;

class ProcessPurchasesss {

   public static void main(String args[]) {
      Scanner myScanner = new Scanner(System.in);
      Purchase aPurchase;

      for (int count = 0; count < 3; count++) {
         aPurchase = new Purchase();
          System.out.print("Amount: ");
         aPurchase.amount = myScanner.nextDouble();
         System.out.print("Taxable? (true/false) ");
         aPurchase.taxable = myScanner.nextBoolean();

         if (aPurchase.taxable) {
            aPurchase.total = aPurchase.amount * 1.05;
         } else {
            aPurchase.total = aPurchase.amount;
         }

         System.out.print("Total: ");
         System.out.println(aPurchase.total);
         System.out.println();
      }
   }
}
```

Figure 17-5 has a run of the code in Listing 17-5, and Figure 17-6 illustrates the concept.

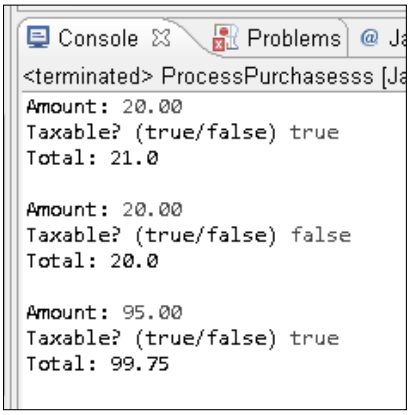


Figure 17-5: Running the code in Listing 17-5.

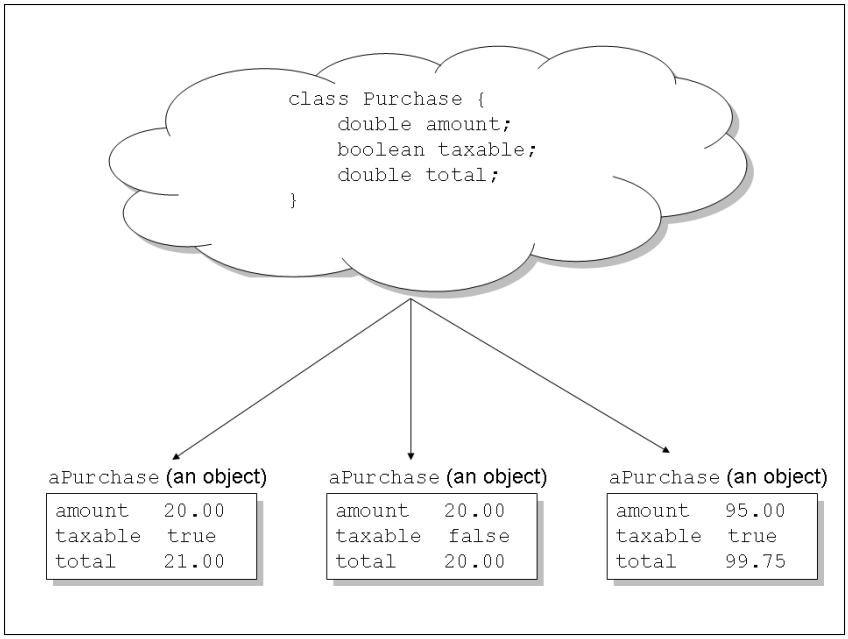


Figure 17-6: From one class come three objects.

To compile the code in Listing 17-5, you must have a copy of the `Purchase` class in the same project. (The `Purchase` class is in Listing 17-2.) To copy a class's code from one project to another, see Chapter 16. (One of that chapter's sidebars describes the copy-and-paste routine.)

Listing 17-5 has only one variable that refers to purchase objects. (The variable's name is `aPurchase`.) The program has three purchase objects because the assignment statement

```
aPurchase = new Purchase();
```

is executed three times (once for each iteration of the `for` loop). Just as you can separate an `int` variable's assignment from the variable's declaration

```
int count;
count = 0;
```

you can also separate a `Purchase` variable's assignment from the variable's declaration:

```
Purchase aPurchase;

for (int count = 0; count < 3; count++) {
    aPurchase = new Purchase();
```

In fact, after you've created the code in Listing 17-2, the word `Purchase` stands for a brand-new type — a reference type. Java has eight built-in primitive types and has as many reference types as people can define during your lifetime. In Listing 17-2, I define the `Purchase` reference type, and you can define reference types, too.

Table 17-2 has a brief comparison of primitive types and reference types.

Table 17-2 Java Types

	Primitive Type	*Reference Type*
How it's created	Built into the language	Defined as a Java class
How many are there	Eight	Indefinitely many
Sample variable declaration	`int count;`	`Purchase aPurchase;`
Sample assignment	`count = 0;`	`aPurchase = new Purchase();`
Assigning a value to one of its parts	(Not applicable)	`aPurchase.amount = 20.00;`

Another Way to Think about Classes

When you start learning object-oriented programming, you may think this class idea is a big hoax. Some geeks in Silicon Valley had nothing better to do, so they went to a bar and made up some confusing gibberish about classes. They don't know what it means, but they have fun watching people struggle to understand it.

Well, that's not what classes are all about. Classes are serious stuff. What's more, classes are useful. Many reputable studies have shown that classes and object-oriented programming save time and money.

Even so, the notion of a class can be very elusive. Even experienced programmers — the ones who are new to object-oriented programming — have trouble understanding how an object differs from a class.

Classes, objects, and tables

Because classes can be so mysterious, I'll expand your understanding with another analogy. Figure 17-7 has a table of three purchases. The table's title

consists of one word (the word "Purchase"), and the table has three column headings — the words "amount," "taxable," and "total." Well, the code in Listing 17-2 has the same stuff — `Purchase`, `amount`, `taxable`, and `total`. So in Figure 17-7, think of the top part of the table (the title and column headings) as a class. Like the code in Listing 17-2, this top part of the table tells us what it means to be a `Purchase`. (It means having an `amount` value, a `taxable` value, and a `total` value.)

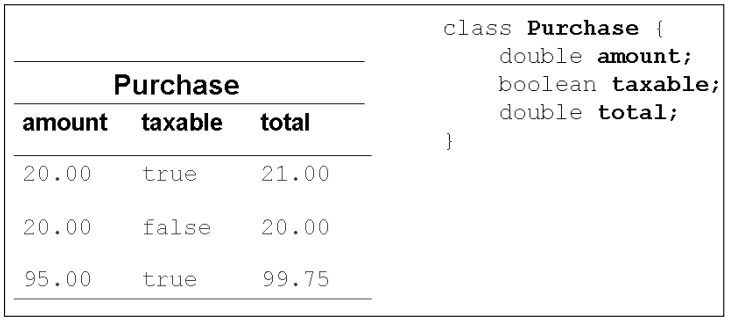


Figure 17-7: A table of purchases.

A class is like the top part of a table. And what about an object? Well, an object is like a row of a table. For example, with the code in Listing 17-5 and the input in Figure 17-5, I create three objects (three instances of the `Purchase` class). The first object has `amount` value 20.00, `taxable` value true, and `total` value 21.00. In the table, the first row has these three values — 20.00, true, and 21.00, as shown in Figure 17-8.

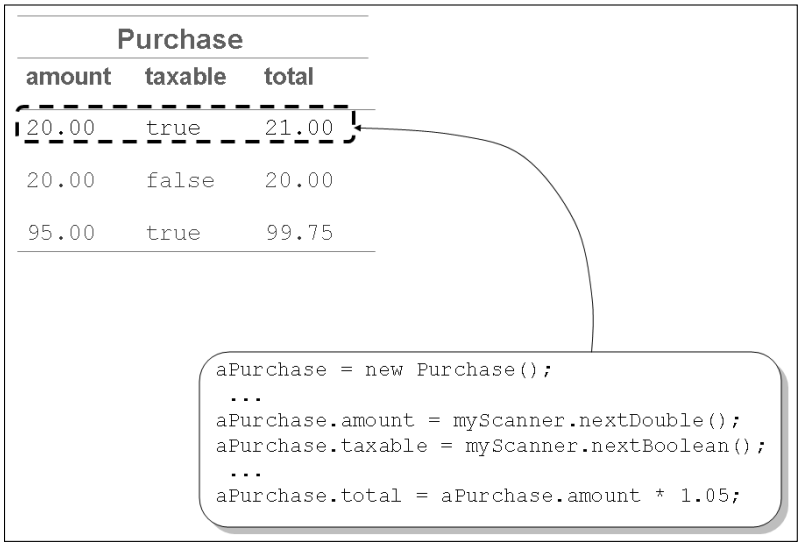


Figure 17-8: A purchase corresponds to a row of the table.

Some questions and answers

Here's the world's briefest object-oriented programming FAQ:

- **Can I have an object without having a class?**

  No, you can't. In Java, every object is an instance of a class.

- **Can I have a class without having an object?**

  Yes, you can. In fact, almost every program in this book creates a class without an object. Take Listing 17-5, for example. The code in Listing 17-5 defines a class named `ProcessPurchasesss`. And nowhere in Listing 17-5 (or anywhere else) do I create an instance of the `ProcessPurchasesss` class. I have a class with no objects. That's just fine. It's business as usual.

- **After I've created a class and its instances, can I add more instances to the class?**

  Yes, you can. In Listing 17-5, I create one instance, then another, and then a third. If I went one additional time around the `for` loop, I'd have a fourth instance, and I'd put a fourth row in the table of Figure 17-8. With no objects, three objects, four objects, or more objects, I still have the same old `Purchase` class.

- **Can an object come from more than one class?**

  Bite your tongue! Maybe other object-oriented languages allow this nasty class cross-breeding, but in Java, it's strictly forbidden. That's one of the things that distinguishes Java from some of the languages that preceded it. Java is cleaner, more uniform, and easier to understand.

Chapter 18

Using Methods and Variables from a Java Class

In This Chapter

- Using Java's String class
- Calling methods
- Understanding static and non-static methods and variables
- Making numbers look good

I hope you didn't read Chapter 17 because I tell a big lie in the beginning of the chapter. Actually, it's not a lie. It's an exaggeration.

Actually, it's not an exaggeration. It's a careful choice of wording. In Chapter 17, I write that the gathering of data into a class is the start of object-oriented programming. Well, that's true. Except that many programming languages had data-gathering features before object-oriented programming became popular. Pascal had *records.* C had *structs.*

To be painfully precise, the grouping of data into usable chunks is only a prerequisite to object-oriented programming. You're not really doing object-oriented programming until you combine both data and methods in your classes.

This chapter starts the "data and methods" ball rolling, and Chapter 19 rounds out the picture.

The String Class

The `String` class is declared in the Java API. This means that, somewhere in the stuff you download from `java.com` is a file named `String.java`. If you hunt down this `String.java` file and peek at the file's code, you find some very familiar-looking stuff:

```
class String {
    ...And so on.
```

In your own code, you can use this `String` class without ever seeing what's inside the `String.java` file. That's one of the great things about object-oriented programming.

A simple example

A `String` is bunch of characters. It's like having several `char` values in a row. You can declare a variable to be of type `String` and store several letters in the variable. Listing 18-1 has a tiny example.

Listing 18-1: I'm Repeating Myself Again (Again)

```
import java.util.Scanner;

class JazzyEchoLine {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        String lineIn;

        lineIn = myScanner.nextLine();
        System.out.println(lineIn);
    }
}
```

A run of Listing 18-1 is shown in Figure 18-1. This run bears an uncanny resemblance to runs of Listing 5-1 from Chapter 5. That's because Listing 18-1 is a reprise of the effort in Listing 5-1.

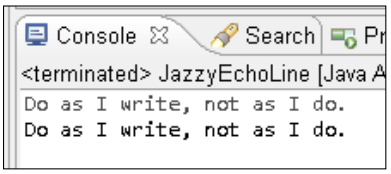


Figure 18-1: Running the code in Listing 18-1.

The new idea in Listing 18-1 is the use of a `String`. In Listing 5-1, I have no variable to store the user's input. But in Listing 18-1, I create the `lineIn` variable. This variable stores a bunch of letters, like the letters `Do as I write, not as I do`.

Putting String variables to good use

The program in Listing 18-1 takes the user's input and echoes it back on the screen. This is a wonderful program, but (like many college administrators that I know) it doesn't seem to be particularly useful.

So take a look at a more useful application of Java's `String` type. A nice one is in Listing 18-2.

Listing 18-2: Putting a Name in a String Variable

```
import java.util.Scanner;
import static java.lang.System.out;

class ProcessMoreData {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        String fullName;
        double amount;
        boolean taxable;
        double total;

        out.print("Customer's full name: ");
        fullName = myScanner.nextLine();
        out.print("Amount: ");
        amount = myScanner.nextDouble();
        out.print("Taxable? (true/false) ");
        taxable = myScanner.nextBoolean();

        if (taxable) {
            total = amount * 1.05;
        } else {
            total = amount;
        }

        out.println();
        out.print("The total for ");
        out.print(fullName);
        out.print(" is ");
        out.print(total);
        out.println(".");
    }
}
```

A run of the code in Listing 18-2 is shown in Figure 18-2. The code stores `Barry A. Burd` in a variable called `fullName` and displays the `fullName` variable's content as part of the output. To make this program work, you

have to store `Barry A. Burd` somewhere. After all, the program follows a certain outline:

```
Get a name.
Get some other stuff.
Compute the total.
Display the name (along with some other stuff).
```

If you don't have the program store the name somewhere then, by the time it's done getting other stuff and computing the total, it forgets the name (so the program can't display the name).

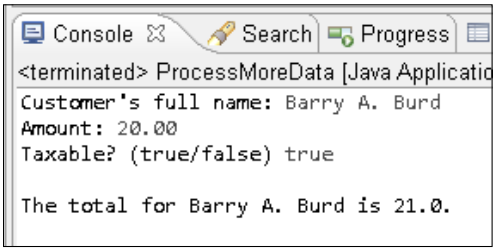


Figure 18-2: Making a purchase.

Reading and writing strings

To read a `String` value from the keyboard, you can call either `next` or `nextLine`:

- **The method `next` reads up to the next blank space.**

  For example, with the input `Barry A. Burd`, the statements

  ```
  String firstName = myScanner.next();
  String middleInit = myScanner.next();
  String lastName = myScanner.next();
  ```

  assign `Barry` to `firstName`, `A.` to `middleInit`, and `Burd` to `lastName`.

- **The method `nextLine` reads up to the end of the current line.**

  For example, with input `Barry A. Burd`, the statement

  ```
  String fullName = myScanner.nextLine();
  ```

  assigns `Barry A. Burd` to the variable `fullName`. (Hey, being an author has some hidden perks.)

To display a `String` value, you can call one of your old friends, `System.out.print` or `System.out.println`. In fact, most of the programs in this book display `String` values. In Listing 18-2, a statement like

```
out.print("Customer's full name: ");
```

displays the `String` value `"Customer's full name: "`.

You can use `print` and `println` to write `String` values to a disk file. For details, see Chapter 13.

Chapter 4 introduces a bunch of characters, enclosed in double quote marks:

```
"Chocolate, royalties, sleep"
```

In Chapter 4, I call this a *literal* of some kind. (It's a literal because, unlike a variable, it looks just like the stuff that it represents.) Well, in this chapter, I can continue the story about Java's literals:

- In Listing 18-2, `amount` and `total` are `double` variables, and `1.05` is a `double` literal.
- In Listing 18-2, `fullName` is a `String` variable, and things like `"Customer's full name: "` are `String` literals.

In a Java program, you surround the letters in a `String` literal with double quote marks.

Using an Object's Methods

If you're not too concerned about classes and reference types, then the use of the type `String` in Listing 18-2 is no big deal. Almost everything you can do with a primitive type seems to work with the `String` type as well. But there's danger around the next curve. Take a look at the code in Listing 18-3 and the run of the code shown in Figure 18-3.

Listing 18-3: A Faulty Password Checker

```
/*
 * This code does not work:
 */
import java.util.Scanner;
import static java.lang.System.out;

class TryToCheckPassword {
```

(continued)

Listing 18-3 *(continued)*

```
    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        String password = "swordfish";
        String userInput;

        out.print("What's the password? ");
        userInput = myScanner.next();

        if (password == userInput) {
            out.println("You're okay!");
        } else {
            out.println("You're a menace.");
        }
    }
}
```

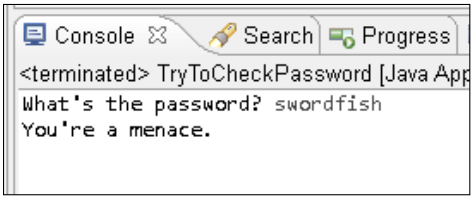


Figure 18-3: But I typed the correct password!

Here are the facts as they appear in this example:

- According to the code in Listing 18-3, the value of `password` is `"swordfish"`.
- In Figure 18-3, in response to the program's prompt, the user types `swordfish`. So in the code, the value of `userInput` is `"swordfish"`.
- The `if` statement checks the condition `password == userInput`. Because both variables have the value `"swordfish"`, the condition *should* be true, but. . . .
- The condition is *not* true because the program's output is `You're a menace.`

What's going on here? I try beefing up the code to see if I can find any clues. An enhanced version of the password-checking program is in Listing 18-4, with a run of the new version shown in Figure 18-4.

Listing 18-4: An Attempt to Debug the Code in Listing 18-3

```
import java.util.Scanner;
import static java.lang.System.out;

class DebugCheckPassword {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        String password = "swordfish";
        String userInput;

        out.print("What's the password? ");
        userInput = myScanner.next();

        out.println();
        out.print("You typed           ");
        out.println(userInput);
        out.print("But the password is ");
        out.println(password);
        out.println();

        if (password == userInput) {
            out.println("You're okay!");
        } else {
            out.println("You're a menace.");
        }
    }
}
```

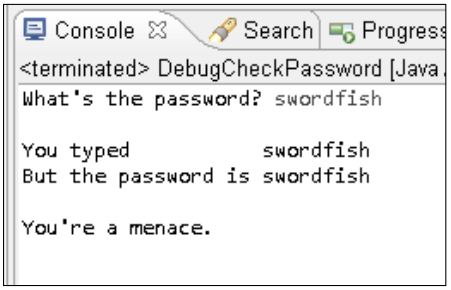


Figure 18-4: This looks even worse.

Ouch! I'm stumped this time. The run in Figure 18-4 shows that both the `userInput` and `password` variables have value `swordfish`. So why doesn't the program accept the user's input?

When you compare two things with a double equal sign, reference types and primitive types don't behave the same way. Consider, for example, `int` versus `String`:

- You can compare two `int` values with a double equal sign. When you do, things work exactly as you would expect. For example, the condition in the following code is true:

```
int apples = 7;
int oranges = 7;

if (apples == oranges) {
   System.out.println("They're equal.");
}
```

- When you compare two `String` values with the double equal sign, things don't work the way you expect. The computer doesn't check to see if the two `String` values contain the same letters. Instead, the computer checks some esoteric property of the way variables are stored in memory.

For your purposes, the term *reference type* is just a fancy name for a class. Because `String` is defined to be a class in the Java API, I call `String` a reference type. This terminology highlights the parallel between primitive types (such as `int`) and classes (that is, reference types, such as `String`).

Comparing strings

In the preceding bullets, the difference between `int` and `String` is mighty interesting. But if the double equal sign doesn't work for `String` values, how do you check to see if Joe User enters the correct password? You do it with the code in Listing 18-5.

Listing 18-5: Calling an Object's Method

```
/*
 * This program works!
 */
import java.util.Scanner;
import static java.lang.System.out;

class CheckPassword {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        String password = "swordfish";
        String userInput;
```

```
        out.print("What's the password? ");
        userInput = myScanner.next();

        if (password.equals(userInput)) {
            out.println("You're okay!");
        } else {

            out.println("You're a menace.");
        }
    }
}
```

A run of the new password-checking code is shown in Figure 18-5 and, let me tell you, it's a big relief! The code in Listing 18-5 actually works! When the user types `swordfish`, the `if` statement's condition is true.

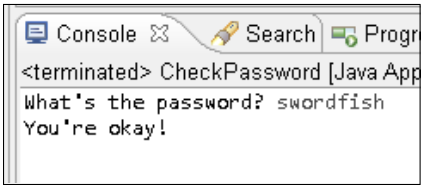


Figure 18-5: At last, Joe User can log in.

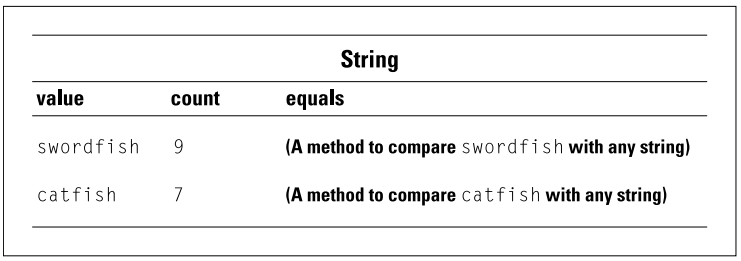
The truth about classes and methods

The magic in Listing 18-5 is the use of a method named `equals`. I have two ways to explain the `equals` method — a simple way, and a more detailed way. First, here's the simple way: The `equals` method compares the characters in one string with the characters in another. If the characters are the same, then the condition inside the `if` statement is true. That's all there is to it.

Don't use a double equal sign to compare two `String` objects. Instead, use one of the objects' `equals` methods.

For a more detailed understanding of the `equals` method, flip to Chapter 17 and take a look at Figures 17-7 and 17-8. Those figures illustrate the similarities between classes, objects, and the parts of a table. In the figures, each row represents a purchase, and each column represents a feature that purchases possess.

You can observe the same similarities for any class, including Java's `String` class. In fact, what Figure 17-7 does for purchases, Figure 18-6 does for strings.



String		
value	**count**	**equals**
swordfish	9	**(A method to compare** swordfish **with any string)**
catfish	7	**(A method to compare** catfish **with any string)**

Figure 18-6: Viewing the String class and String objects as parts of a table.

The stuff shown in Figure 18-6 is much simpler than the real `String` class story. But Figure 18-6 makes a good point. Like the purchases in Figure 17-7, each string has its own features. For example, each string has a `value` (the actual characters stored in the string), and each string has a `count` (the number of characters stored in the string). You can't really write the following line of code:

```
//This code does NOT work:
System.out.println(password.count);
```

but that's because the stuff in Figure 18-6 omits a few subtle details.

Anyway, each row in Figure 18-6 has three items — a `value`, a `count`, and an `equals` method. So each row of the table contains more than just data. Each row contains an `equals` method, a way of doing something useful with the data. It's as if each object (each instance of the `String` class) has three things:

- A bunch of characters (the object's `value`)
- A number (the object's `count`)
- A way of being compared with other strings (the object's `equals` method)

That's the essence of object-oriented programming. Each string has its own personal copy of the `equals` method. For example, in Listing 18-5, the `password` string has its own `equals` method. When you call the `password` string's `equals` method and put the `userInput` string in the method's parentheses, the method compares the two strings to see if those strings contain the same characters.

The `userInput` string in Listing 18-5 has an `equals` method, too. I could use the `userInput` string's `equals` method to compare this string with the `password` string. But I don't. In fact, in Listing 18-5, I don't use the `userInput` string's `equals` method at all. (To compare the `userInput` with the `password`, I had to use either the `password` string's `equals` method or the `userInput` string's `equals` method. So I made an arbitrary choice: I chose the `password` string's method.)

Calling an object's methods

Calling a string's `equals` method is like getting a purchase's `total`. With both `equals` and `total`, you use your old friend, the dot. For example, in Listing 17-3, you write

```
System.out.println(onePurchase.total);
```

and in Listing 18-5, you write

```
if (password.equals(userInput))
```

A dot works the same way for an object's variables and its methods. In either case, a dot takes the object and picks out one of the object's parts. It works whether that part is a piece of data (as in `onePurchase.total`) or a method (as in `password.equals`).

Combining and using data

At this point in the chapter, I can finally say, "I told you so." Here's a quotation from Chapter 17:

> A class is a design plan. The class describes the way in which you intend to *combine* and *use* pieces of data.

A class can define the way you *use* data. How do you use a password and a user's input? You check to see if they're the same. That's why Java's `String` class defines an `equals` method.

An object can be more than just a bunch of data. With object-oriented programming, each object possesses copies of methods for using that object.

Static Methods

You have a fistful of checks. Each check has a number, an amount, and a payee. You print checks like these with your very own laser printer. To print the checks, you use a Java class. Each object made from the `Check` class has three variables (`number`, `amount`, and `payee`). And each object has one method (a `print` method). You can see all this in Figure 18-7.

You'd like to print the checks in numerical order. So you need a method to *sort* the checks. If the checks in Figure 18-7 were sorted, the check with number 1699 would come first, and the check with number 1705 would come last.

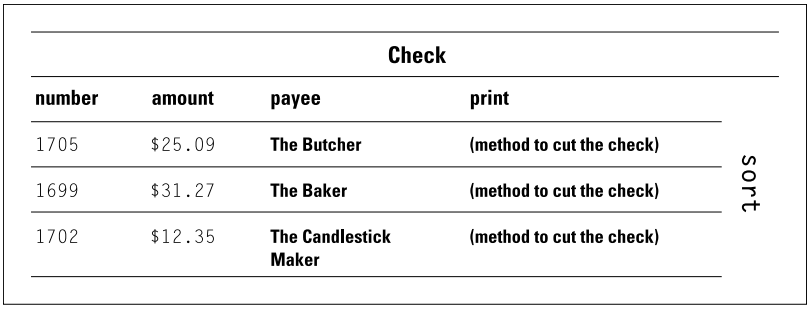


Figure 18-7: The Check class and some check objects.

The big question is, should each check have its own `sort` method? Does the check with number 1699 need to sort itself? And the answer is no. Some methods just shouldn't belong to the objects in a class.

So where do such methods belong? How can you have a `sort` method without creating a separate `sort` for each check?

Here's the answer. You make the `sort` method be *static.* Anything that's static belongs to a whole class, not to any particular instance of the class. If the `sort` method is static, then the entire `Check` class has just one copy of the `sort` method. This copy stays with the entire `Check` class. No matter how many instances of the `Check` class you create — three, ten, or none — you have just one `sort` method.

For an illustration of this concept, refer to Figure 18-7. The whole class has just one `sort` method. So the `sort` method is static. No matter how you call the `sort` method, that method uses the same values to do its work.

Of course, each individual check (each object, each row of the table in Figure 18-7) still has its own `number`, its own `amount`, its own `payee`, and its own `print` method. When you `print` the first check, you get one amount, and when you `print` the second check get another. Because there's a `number`, an `amount`, a `payee`, and a `print` method for each object, I call these things *non-static*. I call them non-static because . . . well . . . because they're not static.

Calling static and non-static methods

In this book, my first use of the word `static` is in Listing 3-1. I use `static` as part of every `main` method (and this book's listings have lots of `main` methods). In Java, your `main` method has to be static. That's just the way it goes.

To call a static method, you use a class's name along with a dot. This is just slightly different from the way you call a non-static method:

- **To call an ordinary (non-static) method, you follow an object with a dot.**

  For example, a program to process the checks in Figure 18-7 may contain code of the following kind:

```
Check firstCheck;
firstCheck.number = 1705;
firstCheck.amount = 25.09;
firstCheck.payee = "The Butcher";
firstCheck.print();
```

- **To call a class's static method, you follow the class name with a dot.**

  For example, to sort the checks in Figure 18-7, you may call

```
Check.sort();
```

Turning strings into numbers

The code in Listing 18-5 introduces a non-static method named `equals`. To compare the `password` string with the `userInput` string, you preface `.equals` with either of the two string objects. In Listing 18-5, I preface `.equals` with the `password` object:

```
if (password.equals(userInput))
```

Each string object has an `equals` method of its own, so I can achieve the same effect by writing

```
if (userInput.equals(password))
```

But Java has another class named `Integer`, and the whole `Integer` class has a static method named `parseInt`. If someone hands you a string of characters, and you want to turn that string into an `int` value, you can call the `Integer` class's `parseInt` method. Listing 18-6 has a small example.

Listing 18-6: More Chips, Please

```
import java.util.Scanner;
import static java.lang.System.out;

class AddChips {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        String reply;
```

(continued)

Listing 18-6 *(continued)*

```
        int numberOfChips;

        out.print("How many chips do you have?");
        out.print(" (Type a number,");
        out.print(" or type 'Not playing') ");
        reply = myScanner.nextLine();

        if (!reply.equals("Not playing")) {
            numberOfChips = Integer.parseInt(reply);
            numberOfChips += 10;

            out.print("You now have ");
            out.print(numberOfChips);
            out.println(" chips.");
        }
    }
}
```

Some runs of the code in Listing 18-6 are shown in Figure 18-8. You want to give each player ten chips. But some party poopers in the room aren't playing. So two people, each with no chips, may not get the same treatment. An empty-handed player gets ten chips, but an empty-handed party pooper gets none.

So in Listing 18-6, you call the `Scanner` class's `nextLine` method, allowing a user to enter any characters at all — not just digits. If the user types `Not playing`, then you don't give the killjoy any chips.

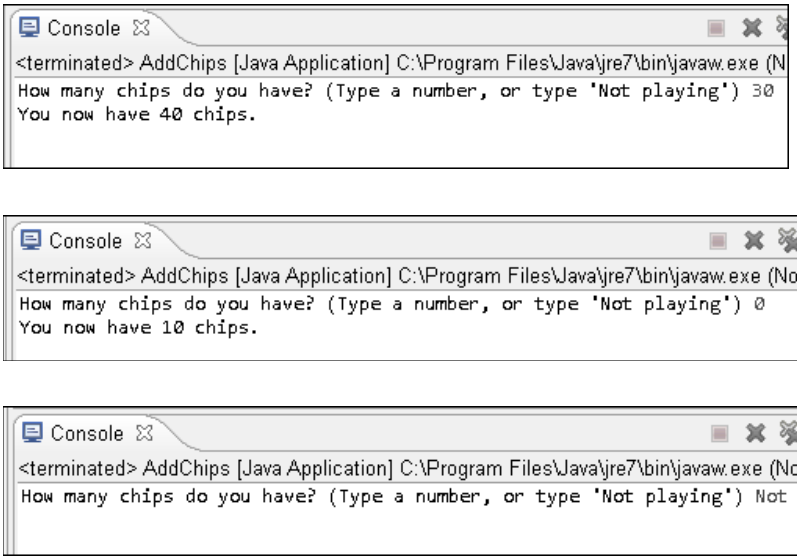

Console
<terminated> AddChips [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (N
How many chips do you have? (Type a number, or type 'Not playing') 30
You now have 40 chips.
Console
<terminated> AddChips [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Nov
How many chips do you have? (Type a number, or type 'Not playing') 0
You now have 10 chips.
Console
<terminated> AddChips [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Nov 9, 2011 1
How many chips do you have? (Type a number, or type 'Not playing') Not playing

Figure 18-8: Running the code in Listing 18-6.

If the user types some digits, then you're stuck holding these digits in the string variable named `reply`. You can't add ten to a string like `reply`. So you call the `Integer` class's `parseInt` method, which takes your string and hands you back a nice `int` value. From there, you can add ten to the `int` value.

Java has a loophole that allows you to add a number to a string. The problem is, you don't get real addition. Adding the number `10` to the string `"30"` gives you `"3010"`, not `40`.

Don't confuse `Integer` with `int`. In Java, `int` is the name of a primitive type (a type that I use throughout this book). But `Integer` is the name of a class. Java's `Integer` class contains handy methods for dealing with `int` values. For example, in Listing 18-6, the `Integer` class's `parseInt` method makes an `int` value from a string.

Turning numbers into strings

In Chapter 17, Listing 17-1 adds tax to the amount of a purchase. But a run of the code in Listing 17-1 has an anomaly. Refer to Figure 17-1. With 5 percent tax on 20 dollars, the program displays a total of `21.0`. That's peculiar. Where I come from, currency amounts aren't normally displayed with just one digit beyond the decimal point.

If you don't choose your purchase amount carefully, the situation is even worse. For example, in Figure 18-9, I run the same program (the code in Listing 17-1) with purchase amount 19.37. The resulting display looks very nasty.

With its internal zeros and ones, the computer doesn't do arithmetic quite the way you and I are used to doing it. So how do you fix this problem?

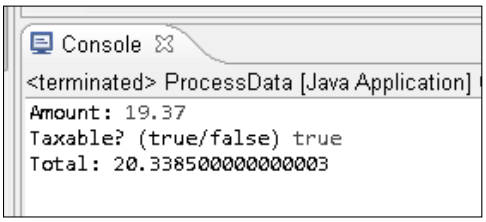


Figure 18-9: Do you have change for 20.338500000000003?

The Java API has a class named `NumberFormat`, and the `NumberFormat` class has a static method named `getCurrencyInstance`. When you call `NumberFormat.getCurrencyInstance()` with nothing inside the parentheses, you get an object that can mold numbers into U.S. currency amounts. Listing 18-7 has an example.

Listing 18-7: The Right Way to Display a Dollar Amount

```
import java.text.NumberFormat;
import java.util.Scanner;

class BetterProcessData {

    public static void main(String args[]) {
        Scanner myScanner = new Scanner(System.in);
        double amount;
        boolean taxable;
        double total;
        NumberFormat currency =
            NumberFormat.getCurrencyInstance();
        String niceTotal;

        System.out.print("Amount: ");
        amount = myScanner.nextDouble();
        System.out.print("Taxable? (true/false) ");
        taxable = myScanner.nextBoolean();

        if (taxable) {
            total = amount * 1.05;
        } else {
            total = amount;
        }

        niceTotal = currency.format(total);
        System.out.print("Total: ");
        System.out.println(niceTotal);
    }
}
```

For some beautiful runs of the code in Listing 18-7, see Figure 18-10. Now at last, you see a total like `$20.34`, not `20.338500000000003`. Ah! That's much better.

How the NumberFormat works

For my current purposes, the code in Listing 18-7 contains three interesting variables:

- The variable `total` stores a number, such as 21.0.
- The variable `currency` stores an object that can mold numbers into U.S. currency amounts.
- The variable `niceTotal` is set up to store a bunch of characters.

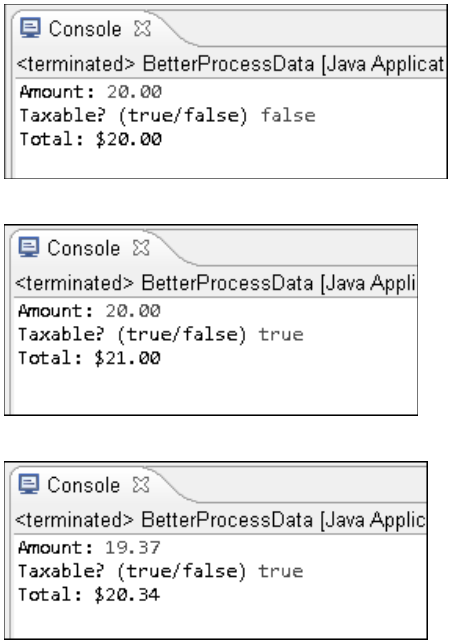


Figure 18-10: See the pretty numbers.

The `currency` object has a `format` method. So to get the appropriate bunch of characters into the `niceTotal` variable, you call the `currency` object's `format` method. You apply this format method to the variable `total`.

Understanding the Big Picture

In this section, I answer some of the burning questions that I raise throughout the book. "What does `java.util` stand for?" "Why do I need the word `static` at certain points in the code?" "How can a degree in Horticultural Studies help you sort cancelled checks?"

I also explain "static" in some unique and interesting ways. After all, static methods and variables aren't easy to understand. It helps to read about Java's static feature from several points of view.

Packages and import declarations

In Java, you can group a bunch of classes into something called a *package.* In fact, the classes in Java's standard API are divided into about 200 packages. This book's examples make heavy use of three packages — the packages named `java.util`, `java.lang`, and `java.io`.

The class java.util.Scanner

The package `java.util` contains about 50 classes, including the very useful `Scanner` class. Like most other classes, this `Scanner` class has two names — a *fully qualified name* and an abbreviated *simple name*. The class's fully qualified name is `java.util.Scanner`, and the class's simple name is `Scanner`. You get the fully qualified name by adding the package name to the class's simple name. (That is, you add the package name `java.util` to the simple name `Scanner`. You get `java.util.Scanner`.)

An import declaration lets you abbreviate a class's name. With the declaration

```
import java.util.Scanner;
```

the Java compiler figures out where to look for the `Scanner` class. So instead of writing `java.util.Scanner` throughout your code, you can just write `Scanner`.

The class java.lang.System

The package `java.lang` contains about 35 classes, including the ever-popular `System` class. (The class's fully qualified name is `java.lang.System`, and the class's simple name is `System`.) Instead of writing `java.lang.System` throughout your code, you can just write `System`. You don't even need an `import` declaration.

All ye need to know

I can summarize much of Java's complexity in only a few sentences:

- The Java API contains many packages.
- A package contains classes.
- From a class, you can create objects.
- An object can have its own methods. An object can also have its own variables.
- A class can have its own static methods. A class can also have its own static variables.

Among all of Java's packages, the `java.lang` package is special. With or without an import declaration, the compiler imports everything in the `java.lang` package. You can start your program with `import java.lang.System`. But if you don't, the compiler adds this declaration automatically.

The static System.out variable

What kind of importing must you do in order to abbreviate `System.out.println`? How can you shorten it to `out.println`? An import declaration lets you abbreviate a *class's* name. But in the expression `System.out`, the word `out` isn't a class. The word `out` is a static variable. (The `out` variable refers to the place where a Java program sends text output.) So you can't write

```
//This code is bogus. Don't use it:
import java.lang.System.out;
```

What do you do instead? You write

```
import static java.lang.System.out;
```

To find out more about the `out` variable's being a static variable, read the next section.

Shedding light on the static darkness

I love to quote myself. When I quote my own words, I don't need written permission. I don't have to think about copyright infringement, and I never hear from lawyers. Best of all, I can change and distort anything I say. When I paraphrase my own ideas, I can't be misquoted.

With that in mind, here's a quote from the previous section:

> *"Anything that's static belongs to a whole class, not to any particular instance of the class. . . . To call a static method, you use a class's name along with a dot."*

How profound! In Listing 18-6, I introduce a static method named `parseInt`. Here's the same quotation applied to the static `parseInt` method:

> *The static `parseInt` method belongs to the whole `Integer` class, not to any particular instance of the `Integer` class. . . . To call the static `parseInt` method, you use the `Integer` class's name along with a dot. You write something like `Integer.parseInt(reply)`.*

That's very nice! How about the `System.out` business that I introduce in Chapter 3? I can apply my quotation to that, too.

> *The static `out` variable belongs to the whole `System` class, not to any particular instance of the `System` class. . . . To refer to the static `out` variable, you use the `System` class's name along with a dot. You write something like `System.out.println()`.*

If you think about what `System.out` means, this static business makes sense. After all, the name `System.out` refers to the place where a Java program sends text output. (When you use Eclipse, the name `System.out` refers to Eclipse's Console view.) A typical program has only one place to send its text output. So a Java program has only one `out` variable. No matter how many objects you create — three, ten, or none — you have just one `out` variable. And when you make something static, you ensure that the program has only one of those things.

All right, then! The `out` variable is static.

To abbreviate the name of a static variable (or a static method), you don't use an ordinary import declaration. Instead, you use a static import declaration. That's why, in Chapter 9 and beyond, I use the word `static` to import the `out` variable:

```
import static java.lang.System.out;
```

Barry makes good on an age-old promise

In Chapter 6, I pull a variable declaration outside of a `main` method. I go from code of the kind in Listing 18-8 to code of the kind that's in Listing 18-9.

Listing 18-8: Declaring a Variable Inside the main Method

```
class SnitSoft {

    public static void main(String args[]) {
        double amount = 5.95;

        amount = amount + 25.00;
        System.out.println(amount);
    }
}
```

Listing 18-9: Pulling a Variable Outside of the main Method

```
class SnitSoft {
    static double amount = 5.95;

    public static void main(String args[]) {
        amount = amount + 25.00;
        System.out.println(amount);
    }
}
```

In Chapter 6, I promise to explain why Listing 18-9 needs the extra word `static` (in `static double amount = 5.95`). Well, with all the fuss about static methods in this chapter, I can finally explain everything.

Refer to Figure 18-7. In that figure, you have checks, and you have a `sort` method. Each individual check has its own `number`, its own `amount`, and its own `payee`. But the entire `Check` class has just one `sort` method.

I don't know about you, but to sort my cancelled checks, I hang them on my exotic Yucca Elephantipes tree. I fasten the higher numbered checks to the upper leaves and put the lower numbered checks on the lower leaves. When I find a check whose number comes between two other checks, I select a free leaf (one that's between the upper and lower leaves).

A program to mimic my sorting method looks something like this:

```
class Check {
    int number;
    double amount;
    String payee;

    static void sort() {
        Yucca tree;

        if (myCheck.number > 1700) {
            tree.attachHigh(myCheck);
        }
        // ... etc.
    }
}
```

Because of the word `static`, the `Check` class has only one `sort` method. And because I declare the `tree` variable inside the static `sort` method, this program has only one `tree` variable. (Indeed, I hang all my cancelled checks on just one Yucca tree.) I can move the `tree` variable's declaration outside of the sort method. But if I do, I may have too many Yucca trees.

```
class Check {
    int number;
    double amount;
    String payee;
    Yucca tree;   //This is bad!
                  //Each check has its own tree.

    static void sort() {
        if (myCheck.number > 5000) {
            tree.attachHigh(myCheck);
        }
        // ... etc.
    }
}
```

In this nasty code, each check has its own number, its own amount, its own payee, and its own tree. But that's ridiculous! I don't want to fasten each check to its own Yucca tree. Everybody knows you're supposed to sort checks with just one Yucca tree. (That's the way the big banks do it.)

When I move the `tree` variable's declaration outside of the `sort` method, I want to preserve the fact that I have only one tree. (To be more precise, I have only one tree for the entire `Check` class.) To make sure that I have only one tree, I declare the `tree` variable to be static.

```
class Check {
    int number;
    double amount;
    String payee;
    static Yucca tree;   //That's better!

    static void sort() {
        if (myCheck.number > 5000) {
            tree.attachHigh(myCheck);
        }
        // ... etc.
    }
}
```

For exactly the same reason, I write **`static`** `double amount` when I move from Listing 18-8 to 18-9.

To find out more about sorting, read *UNIX For Dummies: Quick Reference*, 5th Edition, by Margaret Levine Young and John R. Levine. To learn more about bank checks, read *Managing Your Money Online For Dummies* by Kathleen Sindell. To learn more about trees, read *Landscaping For Dummies* by Phillip Giroux, Bob Beckstrom, and Lance Walheim.

Chapter 19

Creating New Java Methods

In This Chapter

- Writing methods that work with existing values
- Building methods that modify existing values
- Making methods that return new values

In Chapters 3 and 4, I introduce Java methods. I show you how to create a `main` method and how to call the `System.out.println` method. Between that chapter and this one, I make very little noise about methods. In Chapter 18, I introduce a bunch of new methods for you to call, but that's only half of the story.

This chapter completes the circle. In this chapter, you create your own Java methods — not the tired old `main` method that you've been using all along, but some new, powerful Java methods.

Defining a Method within a Class

In Chapter 18, Figure 18-6 introduces an interesting notion — a notion that's at the core of object-oriented programming. Each Java string has its own `equals` method. That is, each string has, built within it, the functionality to compare itself with other strings. That's an important point. When you do object-oriented programming, you bundle data and functionality into a lump called a class. Just remember Barry's immortal words from Chapter 17:

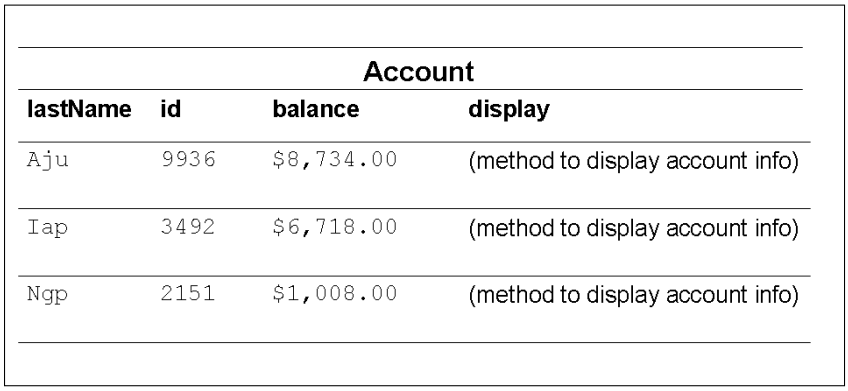
> A class describes the way in which you intend to combine *and use* pieces of data.

And why are these words so important? They're important because in object-oriented programming, chunks of data take responsibility for themselves. With object-oriented programming, everything you have to know about a string is located in the file `String.java`. So if anybody has problems with the strings, they know just where to look for all the code. That's great!

So this is the deal — objects contain methods. Chapter 18 shows you how to use an object's methods, and this chapter shows you how to create an object's methods.

Making a method

Imagine a table containing the information about three accounts. (If you have trouble imagining such a thing, just look at Figure 19-1.) In the figure, each account has a last name, an identification number, and a balance. In addition (and here's the important part), each account knows how to display itself on the screen. Each row of the table has its own copy of a `display` method.



Account			
lastName	**id**	**balance**	**display**
Aju	9936	$8,734.00	(method to display account info)
Iap	3492	$6,718.00	(method to display account info)
Ngp	2151	$1,008.00	(method to display account info)

Figure 19-1: A table of accounts.

The last names in Figure 19-1 may seem strange to you. That's because I generated the table's data randomly. Each last name is a haphazard combination of three letters — one uppercase letter followed by two lowercase letters.

Though it may seem strange, generating account values at random is common practice. When you write new code, you want to test the code to find out if it runs correctly. You can make up your own data (with values like `"Smith"`, `0000`, and `1000.00`). But to give your code a challenging workout, you should use some unexpected values. If you have values from some real-life case studies, you should use them. But if you have don't have real data, randomly generated values are easy to create.

To find out how I randomly generate three-letter names, see this chapter's "Generating words randomly" sidebar.

I need some code to implement the ideas in Figure 19-1. Fortunately, I have some code in Listing 19-1.

Listing 19-1: An Account Class

```
import java.text.NumberFormat;
import static java.lang.System.out;

class Account {
    String lastName;
    int id;
    double balance;

    void display() {
        NumberFormat currency =
            NumberFormat.getCurrencyInstance();

        out.print("The account with last name ");
        out.print(lastName);
        out.print(" and ID number ");
        out.print(id);
        out.print(" has balance ");
        out.println(currency.format(balance));
    }
}
```

The `Account` class in Listing 19-1 defines four things — a `lastName`, an `id`, a `balance`, and a `display`. So each instance of `Account` class has its own `lastName` variable, its own `id` variable, its own `balance` variable, and its own `display` method. These things match up with the four columns in Figure 19-1.

Examining the method's header

Listing 19-1 contains the `display` method's declaration. Like a `main` method's declaration, the `display` declaration has a header and a body (see Chapter 4). The header has two words and some parentheses:

- **The word `void` tells the computer that, when the `display` method is called, the `display` method doesn't return anything to the place that called it.**

  Later in this chapter, a method does return something. For now, the `display` method returns nothing.

- **The word `display` is the method's name.**

  Every method must have a name. Otherwise, you don't have a way to call the method.

- **The parentheses contain all the things you're going to pass to the method when you call it.**

When you call a method, you can pass information to that method on the fly. This `display` example, with its empty parentheses, looks strange. That's because no information is passed to the `display` method when you call it. That's okay. I give a meatier example later in this chapter.

Examining the method's body

The `display` method's body contains some `print` and `println` calls. The interesting thing here is that the body makes reference to the variables `lastName`, `id`, and `balance`. A method's body can do that. But with each object having its own `lastName`, `id`, and `balance` variables, what does a variable in the `display` method's body mean?

Well, when I use the `Account` class, I create little account objects. Maybe I create an object for each row of the table in Figure 19-1. Each object has its own values for the `lastName`, `id`, and `balance` variables, and each object has its own copy of the `display` method.

So take the first `display` method in Figure 19-1 — the method for Aju's account. The `display` method for that object behaves as if it had the code in Listing 19-2.

Listing 19-2: How the display Method Behaves When No One's Looking

```
/*
 * This is not real code:
 */
void display() {
    NumberFormat currency =
        NumberFormat.getCurrencyInstance();

    out.print("The account with last name ");
    out.print("Aju");
    out.print(" and ID number ");
    out.print(9936);
    out.print(" has balance ");
    out.println(currency.format(8734.00));
}
```

In fact, each of the three display methods behaves as if its body has a slightly different code. Figure 19-2 illustrates this idea for two instances of the `Account` class.

Calling the method

To put the previous section's ideas into action, you need more code. So the next listing (see Listing 19-3) creates instances of the `Account` class.

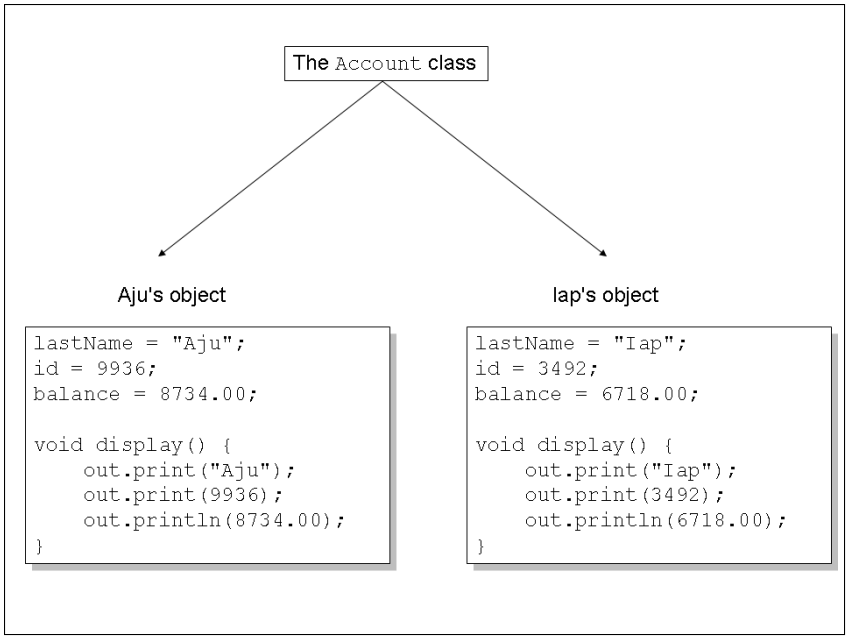


Figure 19-2: Two objects, each with its own display method.

Listing 19-3: Making Use of the Code in Listing 19-1

```
import java.util.Random;

class ProcessAccounts {

    public static void main(String args[]) {

        Random myRandom = new Random();
        Account anAccount;

        for (int i = 0; i < 3; i++) {
            anAccount = new Account();

            anAccount.lastName = "" +
                (char) (myRandom.nextInt(26) + 'A') +
                (char) (myRandom.nextInt(26) + 'a') +
                (char) (myRandom.nextInt(26) + 'a');

            anAccount.id = myRandom.nextInt(10000);
            anAccount.balance = myRandom.nextInt(10000);
            anAccount.display();
        }
    }
}
```

Here's a summary of the action in Listing 19-3:

```
Do the following three times:
    Create a new object (an instance of
        the Account class).
    Randomly generate values for the object's lastName,
        id and balance.
    Call the object's display method.
```

The first of the three `display` calls prints the first object's `lastName`, `id`, and `balance` values. The second `display` call prints the second object's `lastName`, `id`, and `balance` values. And so on.

A run of the code from Listing 19-3 is shown in Figure 19-3.

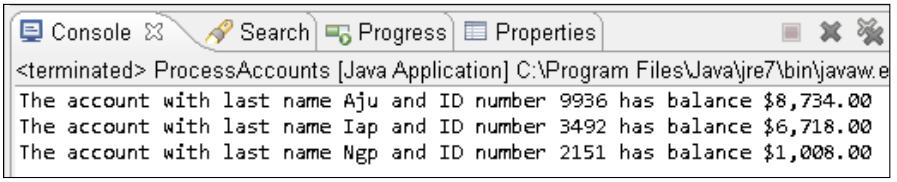

Console
Search
Progress
Properties
<terminated> ProcessAccounts [Java Application] C:\Program Files\Java\jre7\bin\javaw.e
The account with last name Aju and ID number 9936 has balance $8,734.00
The account with last name Iap and ID number 3492 has balance $6,718.00
The account with last name Ngp and ID number 2151 has balance $1,008.00

Figure 19-3: Running the code in Listing 19-3.

Concerning the code in Listing 19-3, your mileage may vary. You don't see the same values as the ones in Figure 19-3. In fact, if you run Listing 19-3 more than once, you (almost certainly) get different three-letter names, different ID numbers, and different account balances each time. That's what happens when a program generates values randomly.

The flow of control

Suppose that you're running the code in Listing 19-3. The computer reaches the `display` method call:

```
anAccount.display();
```

At that point, the computer starts running the code inside the `display` method. In other words, the computer jumps to the middle of the `Account` class's code (the code in Listing 19-1).

After executing the `display` method's code (that forest of `print` and `println` calls), the computer returns to the point where it departed from Listing 19-3. That is, the computer goes back to the `display` method call and continues on from there.

So when you run the code in Listing 19-3, the flow of action in each loop iteration isn't exactly from the top to the bottom. Instead, the action goes from the `for` loop to the `display` method and then back to the `for` loop. The whole business is pictured in Figure 19-4.

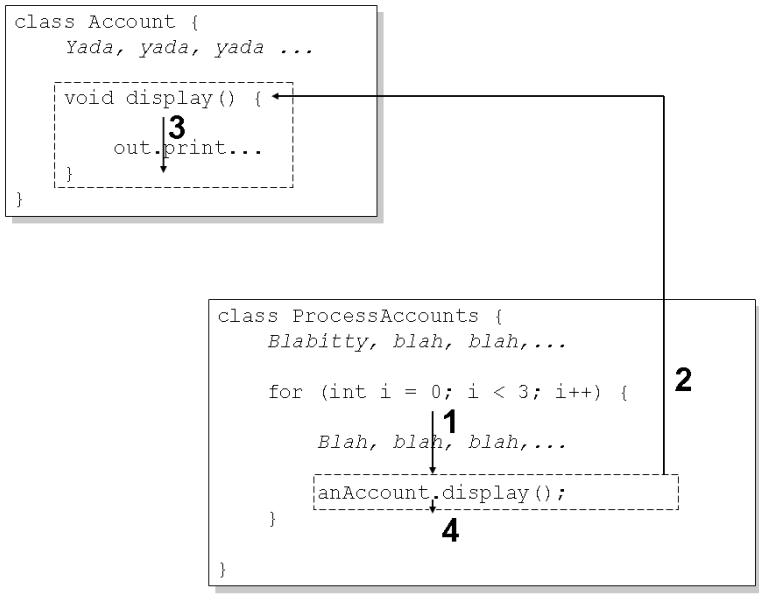


Figure 19-4: The flow of control between Listings 19-1 and 19-3.

Using punctuation

In Listing 19-3, notice the use of dots. To refer to the `lastName` stored in the `anAccount` object, you write

```
anAccount.lastName
```

To get the `anAccount` object to `display` itself, you write

```
anAccount.display();
```

That's great! When you refer to an object's variable or call an object's method, the only difference is parentheses:

- To refer to an object's variable, you don't use parentheses.
- To call an object's method, you use parentheses.

When you call a method, you put parentheses after the method's name. You do this even if you have nothing to put inside the parentheses.

The versatile plus sign

The program in Listing 19-3 uses some cute tricks. In Java, you can do two different things with a plus sign:

- **You can add numbers with a plus sign.**

  For example, you can write

  ```
  numberOfSheep = 2 + 5;
  ```

- **You can *concatenate* strings with a plus sign.**

  When you concatenate strings, you scrunch them together, one right after another. For example, the expression

  ```
  "Barry" + " " + "Burd"
  ```

  scrunches together `Barry`, a blank space, and `Burd`. The new scrunched-up string is (you guessed it) `Barry Burd`.

In Listing 19-3, the statement

```
anAccount.lastName = "" +
    (char) (myRandom.nextInt(26) + 'A') +
    (char) (myRandom.nextInt(26) + 'a') +
    (char) (myRandom.nextInt(26) + 'a');
```

has many plus signs, and some of the plus signs concatenate things together. The first thing is a mysterious empty string (`""`). This empty string is invisible, so it never gets in the way of your seeing the second, third, and fourth things.

Onto the empty string, the program concatenates a second thing. This second thing is the value of the expression `(char) (myRandom.nextInt(26) + 'A')`. The expression may look complicated, but it's really no big deal. This expression represents an uppercase letter (any uppercase letter, generated randomly).

Onto the empty string and the uppercase letter, the program concatenates a third thing. This third thing is the value of the expression `(char) (myRandom.nextInt(26) + 'a')`. This expression represents a lowercase letter (any lowercase letter, generated randomly).

Onto all this stuff, the program concatenates another lowercase letter. So altogether, you have a randomly generated three-letter name. For more details, see the sidebar.

In Listing 19-3, the statement `anAccount.balance = myRandom.nextInt(10000)` assigns an `int` value to `balance`. But `balance` is a `double` variable, not an `int` variable. That's okay. In a rare case of permissiveness, Java allows you to assign an `int` value to a `double` variable. The result of the assignment is no big surprise. If you assign the `int` value 8734 to the `double` variable `balance`, then the value of `balance` becomes 8734.00. The result is shown on the first line of Figure 19-3.

Using the `double` type to store an amount of money is generally a bad idea. In this book, I use `double` to keep the examples as simple as possible. But the `int` type is better for money values, and the `BigDecimal` type is even better. For more details, see Chapter 7.

Generating words randomly

Most programs don't work correctly the first time you run them, and some programs don't work without extensive trial and error. This section's code is a case in point.

To write this section's code, I needed a way to generate three-letter words randomly. After about a dozen attempts, I got the code to work. But I didn't stop there. I kept working for a few hours looking for a *simple* way to generate three-letter words randomly. In the end, I settled on the following code (in Listing 19-3):

```
anAccount.lastName = "" +
(char)
(myRandom.nextInt(26) + 'A') +
(char)
(myRandom.nextInt(26) + 'a') +
(char)
(myRandom.nextInt(26) + 'a');
```

This code isn't simple, but it's not nearly as bad as my original working version. Anyway, here's how the code works:

- **Each call to `myRandom.nextInt(26)` generates a number from 0 to 25.**
- **Adding `'A'` gives you a number from 65 to 90.**

  To store a letter `'A'`, the computer puts the number 65 in its memory. That's why adding `'A'` to 0 gives you 65 and why adding `'A'` to 25 gives you 90. For more information on letters being stored as numbers, see the discussion of Unicode characters at the end of Chapter 8.

- **Applying `(char)` to a number turns the number into a `char` value.**

  To store the letters `'A'` through `'Z'`, the computer puts the numbers 65 through 90 in its memory. So applying `(char)` to a number from 65 to 90 turns the number into an uppercase letter. For more information about applying things like `(char)`, see the discussion of casting in Chapter 7.

(continued)

(continued)

Pause for a brief summary. The expression `(char) (myRandom.nextInt(26) + 'A')` represents a randomly generated uppercase letter. In a similar way, `(char) (myRandom.nextInt(26) + 'a')` represents a randomly generated lowercase letter.

Watch out! The next couple of steps can be tricky.

- **Java doesn't allow you to assign a `char` value to a string variable.**

  So in Listing 19-3, the following statement would lead to a compiler error:

```
//Bad statement:
anAccount.lastName = (char)
(myRandom.nextInt(26) + 'A');
```

- **In Java, you can use a plus sign to add a `char` value to a string. When you do, the result is a string.**

  So `"" + (char) (myRandom.nextInt(26) + 'A')` is string containing one randomly generated uppercase character. And when you add `(char) (myRandom.nextInt(26) + 'a')` onto the end of that string, you get another string — a string containing two randomly generated characters. Finally, when you add another `(char) (myRandom.nextInt(26) + 'a')` onto the end of that string, you get a string containing three randomly generated characters. So you can assign that big string to `anAccount.lastName`. That's how the statement in Listing 19-3 works.

When you write a program like the one in Listing 19-3, you have to be very careful with numbers, `char` values, and strings. I don't do this kind of programming every day of the week. So before I got this section's example to work, I had many false starts. That's okay. I'm very persistent.

Let the Objects Do the Work

When I was a young object, I wasn't as smart as the objects you have nowadays. Consider, for example, the object in Listing 19-4. Not only does this object display itself, but the object can also fill itself with values.

Listing 19-4: A Class with Two Methods

```
import java.util.Random;
import java.text.NumberFormat;
import static java.lang.System.out;

class BetterAccount {
    String lastName;
    int id;
    double balance;
```

```
    void fillWithData() {
        Random myRandom = new Random();

        lastName = "" +
            (char) (myRandom.nextInt(26) + 'A') +
            (char) (myRandom.nextInt(26) + 'a') +
            (char) (myRandom.nextInt(26) + 'a');

        id = myRandom.nextInt(10000);
        balance = myRandom.nextInt(10000);
    }


    void display() {
        NumberFormat currency =

            NumberFormat.getCurrencyInstance();

        out.print("The account with last name ");
        out.print(lastName);
        out.print(" and ID number ");
        out.print(id);
        out.print(" has balance ");
        out.println(currency.format(balance));
    }
}
```

I wrote some code to use the class in Listing 19-4. This new code is in Listing 19-5.

Listing 19-5: This Is So Cool!

```
class ProcessBetterAccounts {

    public static void main(String args[]) {

        BetterAccount anAccount;

        for (int i = 0; i < 3; i++) {
            anAccount = new BetterAccount();
            anAccount.fillWithData();
            anAccount.display();
        }
    }
}
```

Listing 19-5 is pretty slick. Because the code in Listing 19-4 is so darn smart, the new code in Listing 19-5 has very little work to do. This new code just creates a `BetterAccount` object and then calls the methods in Listing 19-4. When you run all this stuff, you get results like the ones in Figure 19-3.

Passing Values to Methods

Think about sending someone to the supermarket to buy bread. When you do this, you say, "Go to the supermarket and buy some bread." (Try it at home. You'll have a fresh loaf of bread in no time at all!) Of course, some other time, you send that same person to the supermarket to buy bananas. You say, "Go to the supermarket and buy some bananas." And what's the point of all this? Well, you have a method, and you have some on-the-fly information that you pass to the method when you call it. The method is named "Go to the supermarket and buy some. . . ." The on-the-fly information is either "bread" or "bananas," depending on your culinary needs. In Java, the method calls would look like this:

```
goToTheSupermarketAndBuySome(bread);
goToTheSupermarketAndBuySome(bananas);
```

The things in parentheses are called *parameters* or *parameter lists.* With parameters, your methods become much more versatile. Instead of getting the same thing each time, you can send somebody to the supermarket to buy bread one time, bananas another time, and birdseed the third time. When you call your `goToTheSupermarketAndBuySome` method, you decide right there and then what you're going to ask your pal to buy.

These concepts are made more concrete in Listings 19-6 and 19-7.

Listing 19-6: Adding Interest

```
import java.text.NumberFormat;
import static java.lang.System.out;

class NiceAccount {
    String lastName;
    int id;
    double balance;

    void addInterest(double rate) {
        out.print("Adding ");
        out.print(rate);
        out.println(" percent...");

        balance += balance * (rate / 100.0);
    }


    void display() {
        NumberFormat currency =
            NumberFormat.getCurrencyInstance();
```

```
        out.print("The account with last name ");
        out.print(lastName);
        out.print(" and ID number ");
        out.print(id);
        out.print(" has balance ");
        out.println(currency.format(balance));
    }
}
```

Listing 19-7: Calling the addInterest Method

```
import java.util.Random;

class ProcessNiceAccounts {

    public static void main(String args[]) {
        Random myRandom = new Random();
        NiceAccount anAccount;
        double interestRate;

        for (int i = 0; i < 3; i++) {
            anAccount = new NiceAccount();

            anAccount.lastName = "" +
                (char) (myRandom.nextInt(26) + 'A') +
                (char) (myRandom.nextInt(26) + 'a') +
                (char) (myRandom.nextInt(26) + 'a');
            anAccount.id = myRandom.nextInt(10000);
            anAccount.balance = myRandom.nextInt(10000);

            anAccount.display();

            interestRate = myRandom.nextInt(5);
            anAccount.addInterest(interestRate);

            anAccount.display();
            System.out.println();
        }
    }
}
```

In Listing 19-7, the line

```
anAccount.addInterest(interestRate);
```

plays the same role as the line `goToTheSupermarketAndBuySome(b read)` in my little supermarket example. The word `addInterest` is a method name, and the word `interestRate` in parentheses is a parameter.

Taken as a whole, this statement tells the code in Listing 19-6 to execute its `addInterest` method. This statement also tells Listing 19-6 to use a certain number (whatever value is stored in the `interestRate` variable) in the method's calculations. The value of `interestRate` can be 1.0, 2.0, or whatever other value you get by calling `myRandom.nextInt(5)`. In the same way, the `goToTheSupermarketAndBuySome` method works for bread, bananas, or whatever else you need from the market.

The next section has a detailed description of `addInterest` and its action. In the meantime, a run of the code in Listings 19-6 and 19-7 is shown in Figure 19-5.

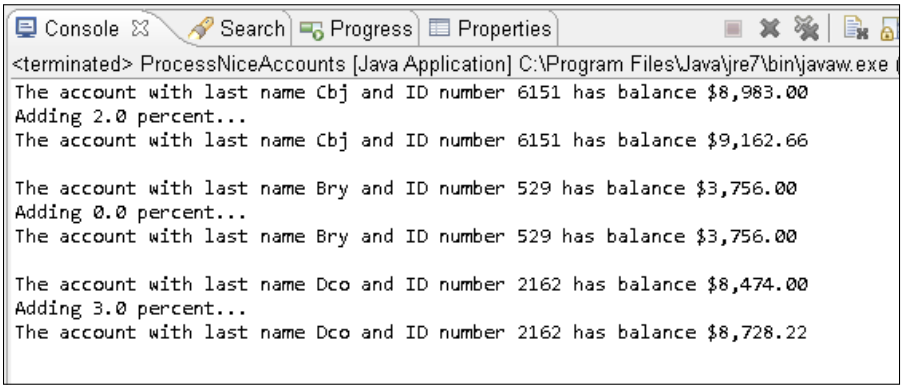

Console
Search
Progress
Properties
<terminated> ProcessNiceAccounts [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe
The account with last name Cbj and ID number 6151 has balance $8,983.00
Adding 2.0 percent...
The account with last name Cbj and ID number 6151 has balance $9,162.66
The account with last name Bry and ID number 529 has balance $3,756.00
Adding 0.0 percent...
The account with last name Bry and ID number 529 has balance $3,756.00
The account with last name Dco and ID number 2162 has balance $8,474.00
Adding 3.0 percent...
The account with last name Dco and ID number 2162 has balance $8,728.22

Figure 19-5: Running the code in Listing 19-7.

Java has very strict rules about the use of types. For example, you can't assign a double value (like 3.14) to an `int` variable. (The compiler simply refuses to chop off the .14 part. You get an error message. So what else is new?) But Java isn't completely unreasonable about the use of types. Java allows you to assign an `int` value (like `myRandom.nextInt(5)`) to a `double` variable (like `interestRate`). If you assign the `int` value 2 to the `double` variable `interestRate`, then the value of `interestRate` becomes 2.0. The result is shown on the second line of Figure 19-5.

Handing off a value

When you call a method, you can pass information to that method on the fly. This information is in the method's parameter list. Listing 19-7 has a call to the `addInterest` method:

```
anAccount.addInterest(interestRate);
```

The first time through the loop, the value of `interestRate` is 2.0. (Remember, I'm using the data in Figure 19-5.) So at that point in the program's run, the method call behaves as if it's the following statement:

```
anAccount.addInterest(2.0);
```

The computer is about to run the code inside the `addInterest` method (a method in Listing 19-6). But first, the computer *passes* the value 2.0 to the parameter in the `addInterest` method's header. So inside the `addInterest` method, the value of `rate` becomes 2.0. For an illustration of this idea, see Figure 19-6.

Here's something interesting. The parameter in the `addInterest` method's header is `rate`. But, inside the `ProcessNiceAccounts` class, the parameter in the method call is `interestRate`. That's okay. In fact, it's standard practice.

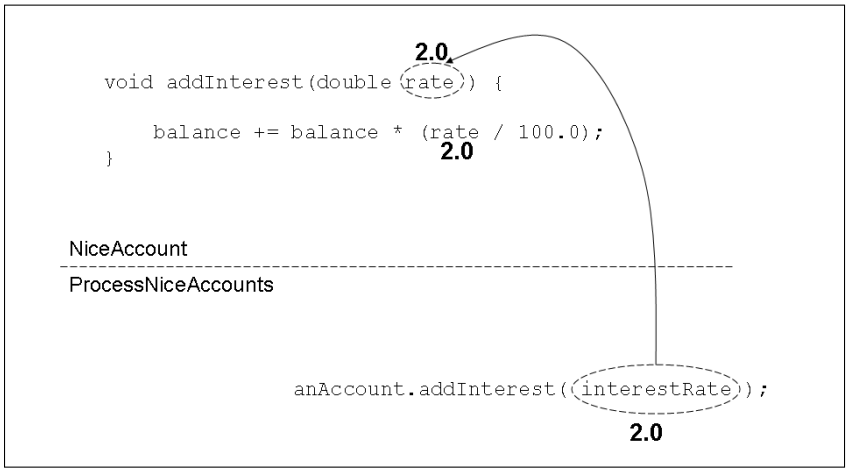


Figure 19-6: Passing a value to a method's parameter.

In Listings 19-6 and 19-7, the names of the parameters don't have to be the same. The only thing that matters is that both parameters (`rate` and `interestRate`) have the same type. In Listings 19-6 and 19-7, both of these parameters are of type `double`. So everything is fine.

Inside the `addInterest` method, the `+=` assignment operator adds `balance * (rate / 100.0)` to the existing `balance` value. For some info about the `+=` assignment operator, see Chapter 7.

Working with a method header

In the next few bullets, I make some observations about the `addInterest` method header (in Listing 19-6):

- **The word `void` tells the computer that when the `addInterest` method is called, the `addInterest` method doesn't send a value back to the place that called it.**

  The next section has an example in which a method sends a value back.

- **The word `addInterest` is the method's name.**

  That's the name you use to call the method when you're writing the code for the `ProcessNiceAccounts` class (see Listing 19-7).

- **The parentheses in the header contain placeholders for all the things you're going to pass to the method when you call it.**

  When you call a method, you can pass information to that method on the fly. This information is the method's parameter list. The `addInterest` method's header says that the `addInterest` method takes one piece of information, and that piece of information must be of type `double`:

  ```
  void addInterest(double rate)
  ```

  Sure enough, if you look at the call to `addInterest` (down in the `ProcessNiceAccounts` class's `main` method), that call has the variable `interestRate` in it. And `interestRate` is of type `double`. When I call `getInterest`, I'm giving the method a value of type `double`.

How the method uses the object's values

The `addInterest` method in Listing 19-6 is called three times from the `main` method in Listing 19-7. The actual account balances and interest rates are different each time:

- **In the first call of Figure 19-5, the `balance` is 8983.00, and the interest rate is 2.0.**

  When this call is made, the expression `balance * (rate / 100.0)` stands for 8983.00 * (2.0 / 100.00). See Figure 19-7.

- **In the second call of Figure 19-5, the balance is 3756.00, and the interest rate is 0.0.**

  When the call is made, the expression `balance * (rate / 100.0)` stands for 3756.00 * (0.0 / 100.00). Again, see Figure 19-7.

- **In the third call of Figure 19-5, the balance is 8474.00, and the interest rate is 3.0.**

  When the `addInterest` call is made, the expression `balance * (rate / 100.0)` stands for 8474.00 * (3.0 / 100.00).

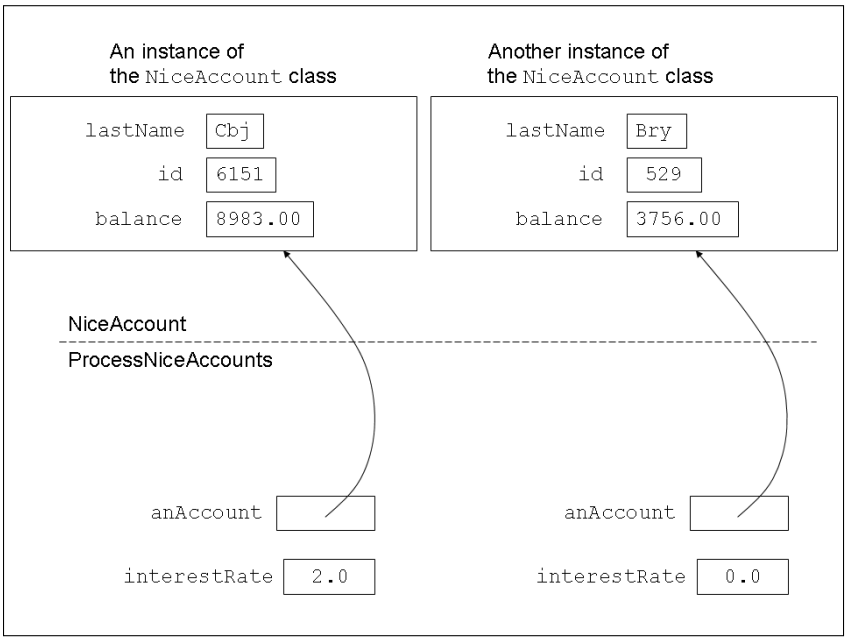


Figure 19-7: Cbj's account and Bry's account.

Getting a Value from a Method

Say that you're sending a friend to buy groceries. You make requests for groceries in the form of method calls. You issue calls such as

```
goToTheSupermarketAndBuySome(bread);
goToTheSupermarketAndBuySome(bananas);
```

The things in parentheses are parameters. Each time you call your `goToTheSupermarketAndBuySome` method, you put a different value in the method's parameter list.

Now what happens when your friend returns from the supermarket? "Here's the bread you asked me to buy," says your friend. As a result of carrying out

your wishes, your friend returns something to you. You made a method call, and the method returns information (or better yet, the method returns some food).

The thing returned to you is called the method's *return value,* and the type of thing returned to you is called the method's *return type.*

An example

To see how return values and a return types work in a real Java program, check out the code in Listings 19-8 and 19-9.

Listing 19-8: A Method That Returns a Value

```
import java.text.NumberFormat;
import static java.lang.System.out;

class GoodAccount {
    String lastName;
    int id;
    double balance;

    double getInterest(double rate) {
        double interest;

        out.print("Adding ");
        out.print(rate);
        out.println(" percent...");

        interest = balance * (rate / 100.0);
        return interest;
    }


    void display() {
        NumberFormat currency =
            NumberFormat.getCurrencyInstance();

        out.print("The account with last name ");
        out.print(lastName);
        out.print(" and ID number ");
        out.print(id);
        out.print(" has balance ");
        out.println(currency.format(balance));
    }
}
```

Listing 19-9: Calling the Method in Listing 19-8

```
import java.util.Random;
import java.text.NumberFormat;

class ProcessGoodAccounts {

    public static void main(String args[]) {
        Random myRandom = new Random();
        NumberFormat currency =
            NumberFormat.getCurrencyInstance();
        GoodAccount anAccount;
        double interestRate;
        double yearlyInterest;
        for (int i = 0; i < 3; i++) {
            anAccount = new GoodAccount();

            anAccount.lastName = "" +
                (char) (myRandom.nextInt(26) + 'A') +
                (char) (myRandom.nextInt(26) + 'a') +
                (char) (myRandom.nextInt(26) + 'a');
            anAccount.id = myRandom.nextInt(10000);
            anAccount.balance = myRandom.nextInt(10000);

            anAccount.display();

            interestRate = myRandom.nextInt(5);
            yearlyInterest =
                anAccount.getInterest(interestRate);

            System.out.print("This year's interest is ");
            System.out.println
                (currency.format(yearlyInterest));
            System.out.println();
        }
    }
}
```

To see a run of code from Listings 19-8 and 19-9, take a look at Figure 19-8.

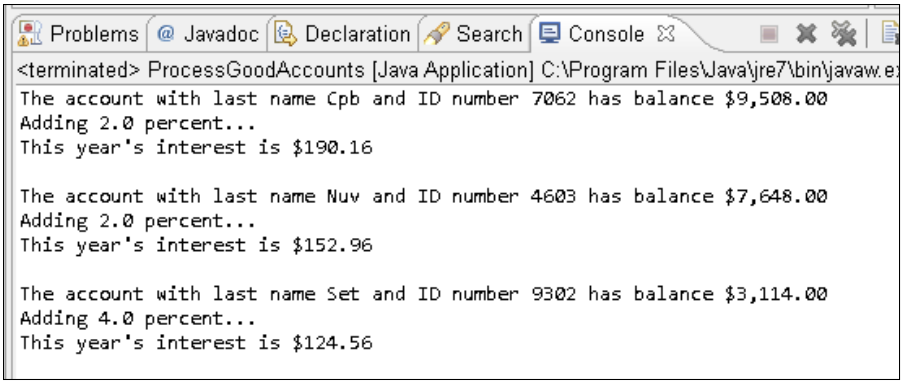

Problems
Javadoc
Declaration
Search
Console
<terminated> ProcessGoodAccounts [Java Application] C:\Program Files\Java\jre7\bin\javaw.e
The account with last name Cpb and ID number 7062 has balance $9,508.00
Adding 2.0 percent...
This year's interest is $190.16
The account with last name Nuv and ID number 4603 has balance $7,648.00
Adding 2.0 percent...
This year's interest is $152.96
The account with last name Set and ID number 9302 has balance $3,114.00
Adding 4.0 percent...
This year's interest is $124.56

Figure 19-8: Running the code in Listing 19-9.

How return types and return values work

I want to trace a piece of the action in Listings 19-8 and 19-9. For input data, I use the first set of values in Figure 19-8.

Here's what happens when `getInterest` is called (you can follow along in Figure 19-9):

- The value of `balance` is 9508.00, and the value of `rate` is 2.0. So the value of `balance * (rate / 100.0)` is 190.16 — one-hundred ninety dollars and sixteen cents.
- The value 190.16 gets assigned to the `interest` variable, so the statement

  ```
  return interest;
  ```

  has the same effect as

  ```
  return 190.16;
  ```

- The `return` statement sends this value 190.16 back to the code that called the method. *At that point in the process, the entire method call in Listing 19-9 —* `anAccount.getInterest(interestRate)` *— takes on the value 190.16.*
- Finally, the value 190.16 gets assigned to the variable `yearlyInterest`.

If a method returns anything, then a call to the method is an expression with a value. That value can be printed, assigned to a variable, added to something else, or whatever. Anything you can do with any other kind of value, you can do with a method call.

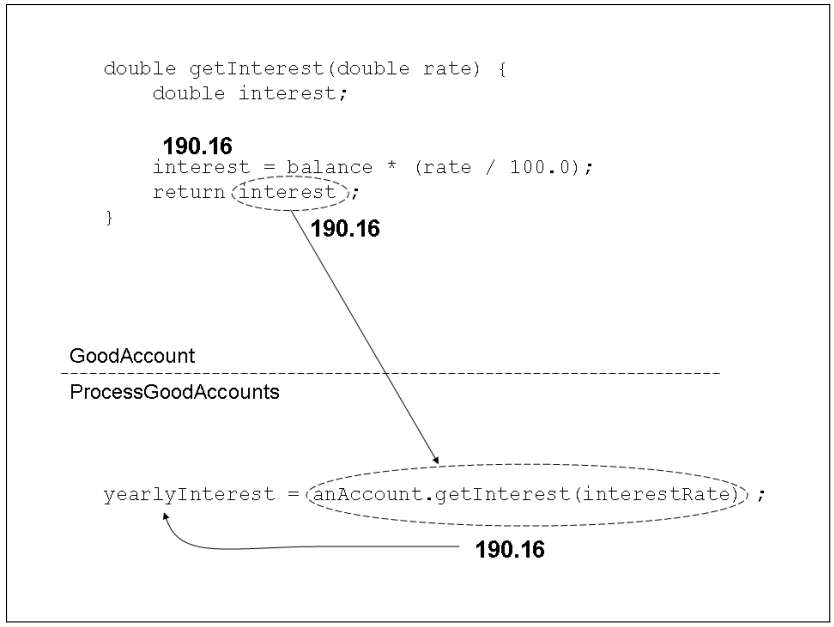


Figure 19-9: A method call is an expression with a value.

Working with the method header (again)

When you create a method or a method call, you have to be careful to use Java's types consistently. So make sure that you check for the following:

- In Listing 19-8, the `getInterest` method's header starts with the word `double`. So when the method is executed, it should send a `double` value back to the place that called it.
- Again in Listing 19-8, the last statement in the `getInterest` method is `return interest`. So the method returns whatever value is stored in the `interest` variable, and the `interest` variable has type `double`. So far, so good.
- In Listing 19-9, the value returned by the call to `getInterest` is assigned to a variable named `yearlyInterest`. Sure enough, `yearlyInterest` is of type `double`.

That settles it! The use of types in the handling of method `getInterest` is consistent in Listings 19-8 and 19-9. I'm thrilled!

Chapter 20

Oooey GUI Was a Worm

In This Chapter

- Swinging into action
- Displaying an image
- Using buttons and text boxes

There's a wonderful old joke about a circus acrobat jumping over mice. Unfortunately, I'd get sued for copyright infringement if I included the joke in this book.

Anyway, the joke is about starting small and working your way up to bigger things. That's what you do when you read *Beginning Programming with Java For Dummies,* 3rd Edition.

Most of the programs in this book are text-based. A *text-based* program has no windows; no dialog boxes; nothing of that kind. With a text-based program, the user types characters in the Console view, and the program displays output in the same Console view.

These days, very few publicly available programs are text-based. Almost all programs use a *GUI* — a *Graphical User Interface.* So if you've read every word of this book up to now, you're probably saying to yourself, "When am I going to find out how to create a GUI?"

Well, now's the time! This chapter introduces you to the world of GUI programming in Java.

The Java Swing Classes

Java's *Swing* classes create graphical objects on a computer screen. The objects can include buttons, icons, text fields, check boxes, and other good things that make windows so useful.

The name "Swing" isn't an acronym. When the people at Sun Microsystems were first creating the code for these classes, one of the developers named it "Swing" because swing music was enjoying a nostalgic revival. And yes, in addition to `String` and `Swing`, the standard Java API has a `Spring` class. But that's another story.

Actually, Java's API has several sets of windowing components. An older set is called *AWT* — the *Abstract Windowing Toolkit.* But to use some of the `Swing` classes, you have to call on some of the old AWT classes. Go figure!

Showing an image on the screen

The program in Listing 20-1 displays a window on your computer screen. To see the window, look at Figure 20-1.

Listing 20-1: Creating a Window with an Image in It

```
import javax.swing.JFrame;
import javax.swing.ImageIcon;
import javax.swing.JLabel;
import java.awt.Container;

class ShowPicture {

    public static void main(String args[]) {
        JFrame frame = new JFrame();
        ImageIcon icon = new ImageIcon("androidBook.jpg");
        JLabel label = new JLabel(icon);

        frame.add(label);
        frame.setDefaultCloseOperation
           (JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```

When you view Listing 20-1 in the Eclipse editor, you see a little yellow marker. A yellow marker represents a warning rather than an error, so you can ignore the warning and still run your code. If you hover over the marker, you see a tip about something called a serialVersionUID. A *serialVersionUID* is a number that helps Java avoid version conflicts when you send different copies of an object from one place to another. You can get rid of the warning by applying one of Eclipse's quick fixes, but if you're not fussy, don't bother with these quick fixes.

The code in Listing 20-1 has very little logic of its own. Instead, this code pulls together a bunch of classes from the Java API.

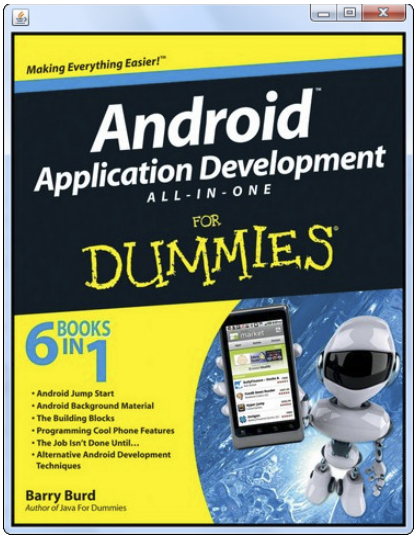


Figure 20-1: What a nice window!

Back in Listing 17-3, I create an instance of the `Purchase` class with the line

```
Purchase onePurchase = new Purchase();
```

So in Listing 20-1, I do the same kind of thing. I create instances of the `JFrame`, `ImageIcon`, and `JLabel` classes with the following lines:

```
JFrame frame = new JFrame();
ImageIcon icon = new ImageIcon("androidBook.jpg");
JLabel label = new JLabel(icon);
```

Here's some gossip about each of these lines:

- A `JFrame` is like a window (except that it's called a `JFrame`, not a "window"). In Listing 20-1, the line

  ```
  JFrame frame = new JFrame();
  ```

  creates a `JFrame` object, but this line doesn't display the `JFrame` object anywhere. (The displaying comes later in the code.)

  In this chapter, I use the words *JFrame, frame, and window* interchangeably. (Well, I use them almost interchangeably. You don't have to think about any of the differences.)

- An `ImageIcon` object is a picture. At the root of the program's project directory, I have a file named `androidBook.jpg`. That file contains the picture shown in Figure 20-1. So in Listing 20-1, the line

  ```
  ImageIcon icon = new ImageIcon("androidBook.jpg");
  ```

creates an `ImageIcon` object — an icon containing the `androidBook.jpg` picture.

For some reason that I'll never understand, you may not want to use my `androidBook.jpg` image file when you run Listing 20-1. You can use almost any `.gif`, `.jpg`, or `.png` file in place of my (lovely) Android book cover image. To do so, drag your own image file to Eclipse's Package Explorer. (Drag it to the root of this example's project folder.) Then, in Eclipse's editor, change the name `androidBook.jpg` to your own image file's name. That's it!

- I need a place to put the icon. I can put it on something called a `JLabel`. So in Listing 20-1, the line

```
JLabel label = new JLabel(icon);
```

creates a `JLabel` object and puts the `androidBook.jpg` icon on the new label's face.

If you read the previous bullets, you may get a false impression. The wording may suggest that the use of each component (`JFrame`, `ImageIcon`, `JLabel`, and so on) is a logical extension of what you already know. "Where do you put an `ImageIcon`? Well of course, you put it on a `JLabel`." When you've worked long and hard with Java's Swing components, all these things become natural to you. But until then, you look everything up in Java's API documentation.

You never need to memorize the names or features of Java's API classes. Instead, you keep Java's API documentation handy. When you need to know about a class, you look it up in the documentation. If you need a certain class often enough, you'll remember its features. For classes that you don't use often, you always have the docs.

For tips on using Java's API documentation, see the Appendix on this book's website — `http://allmycode.com/BeginProg3`. To find gobs of sample Java code, visit some of the websites listed in Chapter 21.

Just another class

What is a `JFrame`? Like any other class, a `JFrame` has several parts. For a simplified view of some of these parts, see Figure 20-2.

Like the `String` in Figure 18-6 in Chapter 18, each object formed from the `JFrame` class has both data parts and method parts. The data parts include the frame's height and width. The method parts include `add`, `setDefaultCloseOperation`, `pack`, and `setVisible`. All told, the `JFrame` class has about 320 methods.

```
public class JFrame {
  int height;
  int width;
  public Component add() ...
  public void setDefaultCloseOperation() ...
  public void pack() ...
  public void setVisible() ...
  ...
}
```

JFrame

height	width	add	setDefaultCloseOperation	pack	setVisible
485	375	(method to place something on the frame)	(method to decide what what happens when the user closes the frame)	(method to shrink-wrap the frame)	(method to make the frame visible or invisible)

Figure 20-2: A simplified depiction of the JFrame class.

For technical reasons too burdensome for this book, you can't use dots to refer to a frame's height or width. But you can call many `JFrame` methods with those infamous dots. In Listing 20-1, I call the frame's methods by writing `add(label)`, `frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE)`, `frame.pack()`, and `frame.setVisible(true)`.

Here's the scoop on the `JFrame` methods in Listing 20-1:

- The call `frame.add(label)` plops the label onto the frame. The label displays my *androidBook.jpg* picture, so this call makes the picture appear on the frame.
- A call to `frame.setDefaultCloseOperation` tells Java what to do when you try to close the frame. (In Windows, you click the 'x' in the upper-right-hand corner by the title bar. On a Mac, the 'x' is in the frame's upper-left corner.) For a frame that's part of a larger application, you may want the frame to disappear when you click the 'x', but you probably don't want the application to stop running.

  But in Listing 20-1, the frame is the entire application — the whole enchilada. So when you click the 'x', you want the Java virtual machine to shut itself down. To make this happen, you call the `setDefaultCloseOperation` method with parameter `JFrame.EXIT_ON_CLOSE`. The other alternatives are as follows:

- `JFrame.HIDE_ON_CLOSE`: The frame disappears, but it still exists in the computer's memory.
- `JFrame.DISPOSE_ON_CLOSE`: The frame disappears and no longer exists in the computer's memory.
- `JFrame.DO_NOTHING_ON_CLOSE`: The frame still appears, still exists, and still does everything it did before you clicked the 'x.' Nothing happens when you click 'x.' So with this `DO_NOTHING_ON_CLOSE` option, you can become very confused.

If you don't call `setDefaultCloseOperation`, then Java automatically chooses the `HIDE_ON_CLOSE` option. When you click the 'x', the frame disappears, but the Java program keeps running. Of course, with no visible frame, the running of Listing 20-1 doesn't do much. The only noticeable effect of the run is your development environment's behavior. With Eclipse, the little square in the Console view's toolbar retains its bright red color. When you hover over the square, you see the Terminate tooltip. So to end the Java program's run (and to return the square to its washed-out reddish-grey hue) simply click this little square.

- A frame's `pack` method shrink-wraps the frame around whatever has been added to the frame's content pane. Without calling `pack`, the frame can be much bigger or much smaller than is necessary.

  Unfortunately, the default is to make a frame much smaller than necessary. If, in Listing 20-1, you forget to call `frame.pack`, you get the tiny frame shown in Figure 20-3. Sure, you can enlarge the frame by dragging the frame's edges with your mouse. But why should you have to do that? Just call `frame.pack` instead.

- Calling `setVisible(true)` makes the frame appear on your screen. If you forget to call `setVisible(true)` (and I often do), when you run the code in Listing 20-1, you'll see nothing on your screen. It's always so disconcerting until you figure out what you did wrong.

Figure 20-3: A frame that hasn't been packed.

Using Eclipse's WindowBuilder

A GUI program is nothing special. Every GUI program is simply a Java source file (or more likely, several Java source files) like the files in this book's code listings. But GUI programs have two interesting characteristics:

- **GUI programs typically contain lots of Java code.**

  Much of this code differs very little from one GUI program to another.

- **GUI programs involve visual elements.**

  The best way to describe visual elements is to "draw" them. Describing them with Java source code can be slow an unintuitive.

So to make your GUI life easier, you can add WindowBuilder to the Eclipse IDE. With WindowBuilder, you describe your program visually. WindowBuilder automatically turns your visual description into real Java source code.

Some websites use the name WindowBuilder Pro. As far as I know, WindowBuilder Pro is another name for plain old WindowBuilder.

Installing WindowBuilder

Eclipse has its own elaborate facility for incorporating new functionality. An Eclipse tool is called an Eclipse *plugin*. When you first install Eclipse, you get many plugins by default. Then, to enhance Eclipse's power, you can install many additional plugins.

Eclipse's WindowBuilder plugin facilitates the creation of GUI applications. Here's how you install WindowBuilder:

1. **In Eclipse's main menu, choose Help➪Install New Software.**

   An Install dialog opens.

2. **In the Install dialog, click the Add button.**

   An Add Repository dialog opens.

3. **In the Add Repository dialog, type a name (hopefully a name that reminds you of WindowBuilder) and a Location URL (see Figure 20-4).**

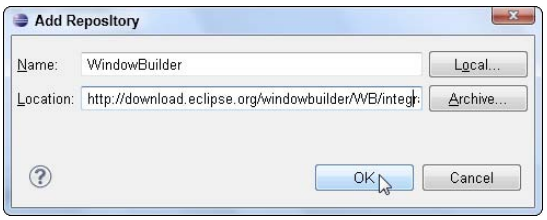


Figure 20-4: Eclipse's Add Repository dialog.

The Location URL points to a place on the web where the WindowBuilder plugin is available for download. In early 2012, the correct URL is `http://download.eclipse.org/windowbuilder/WB/integration/3.8/` and `http://dl.google.com/eclipse/inst/d2wbpro/latest/3.7` works as well. Unfortunately, I can't guarantee that the same URLs will work when you read this book. But if you visit `www.eclipse.org/windowbuilder` in your web browser and do some poking around, you'll probably find the most recent URL. Look for something called the Update Site URL.

The Add Repository dialog's Location field isn't like your web browser's Address field. You can't abbreviate the URL by omitting the `http://` part.

4. **Click OK to close the Add Repository dialog.**

   At this point in your journey, the Install dialog displays a list of plugins that are available on your Location URL's server (see Figure 20-5). If you're like me, you want everything.

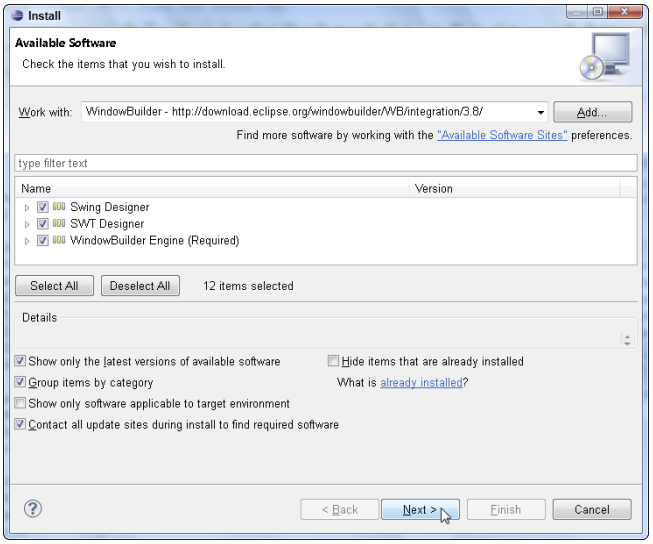


Figure 20-5: The Install dialog.

5. **Click the Select All button and then click Next.**

   At this point, Eclipse asks for your acceptance of the license agreement's terms (which include no mention of your first-born child).

6. **Accept the agreement and follow any other instructions that Eclipse throws at you.**

   After some clicking and agreeing, your download begins. The plugin installs itself automatically into Eclipse. When the download is finished and you restart Eclipse, you have full use of the Eclipse WindowBuilder plugin.

Creating a GUI class

There's a wise old saying, "A picture is worth 70 words." And if you count things like `java.awt.EventQueue` as three separate words, the same picture is worth 80 words. In this section, you create a picture from which Eclipse builds an 80-word Java program.

1. **Follow the instructions in the previous section (the "Installing WindowBuilder" section).**
2. **Create a new Java project.**

   If you're following my instructions to the letter, name the project `WindowBuilderProject`.
3. **Select the new project's branch in Eclipse's Package Explorer.**
4. **In Eclipse's main menu, choose File➪New➪Other.**

   The Select A Wizard dialog box appears.
5. **In the Select A Wizard dialog box's tree, expand the WindowBuilder branch.**
6. **Within the WindowBuilder branch, expand the Swing Designer branch.**

   Java's Swing classes create graphical objects on a computer screen. The objects can include buttons, icons, text fields, check boxes, and other good things that make windows so useful.
7. **Within the Swing Designer branch, double-click the JFrame item.**

   A New JFrame dialog box appears, as shown in Figure 20-6.

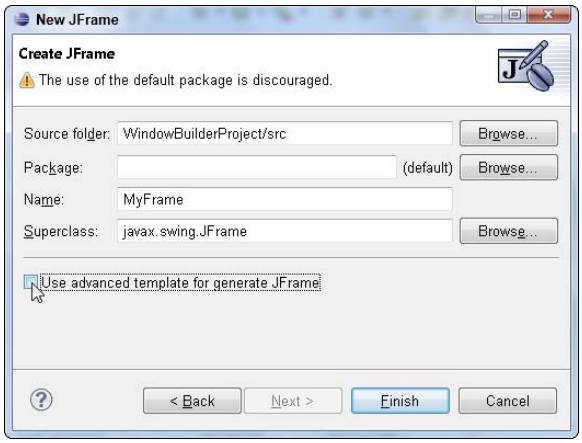


Figure 20-6: The New JFrame dialog box.

8. **In the dialog box's Name field, type a name for your new Java class.**

   If you're following my instructions faithfully, name the class `MyFrame`.

9. **Remove the check mark from the box labeled Use Advanced Template For Generate JFrame.**

   The label's wording is very strange. I didn't create the wording, so don't blame me! If the creators of WindowBuilder have replaced "for generate" with "to generate" by the time you read this book, then remove the check mark from that sensibly labeled check box.

   If you perform Steps 8 and 9 out of order, you may get unexpected results. For some odd reason, Eclipse puts a check mark in the Use Advanced Template check box when you enter anything (even a single character) in the Name field. With any luck, this bizarre behavior will be fixed by the time you install Eclipse.

10. **Click the Finish button to dismiss the New JFrame dialog box.**

    Eclipse's workbench reappears. Your new MyFrame.java file appears in the editor.

Running your bare-bones GUI class

In the previous section, you use WindowBuilder to create a brand-new Java class. When you run the new class's code, you see the stuff in Figure 20-7. You see a window (more precisely, a JFrame instance) with nothing inside it.

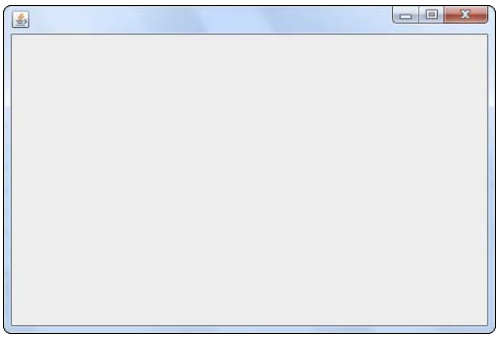


Figure 20-7: An empty JFrame.

The fact that this JFrame contains no images, no buttons, no text fields — no *nothing* — comes from the way WindowBuilder creates your new class. WindowBuilder populates the class with a minimum amount of code. That way, the new class is a blank slate — an empty shell to which you add buttons, text fields, or other useful components.

The next section describes the Java code that creates the display in Figure 20-7.

Show me the code

Listing 20-2 contains the code that WindowBuilder generates automatically. When you run the code, you get the rather bland display in Figure 20-7.

Listing 20-2: WindowBuilder creates a minimal GUI application

```
import java.awt.EventQueue;

import javax.swing.JFrame;


public class MyFrame extends JFrame {

  /**
   * Launch the application.
   */
  public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
      public void run() {
        try {
          MyFrame frame = new MyFrame();
          frame.setVisible(true);
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    });
  }

  /**
   * Create the frame.
   */
  public MyFrame() {
    setBounds(100, 100, 450, 300);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  }

}
```

Listing 20-2 contains several features that tend to confuse new programmers. But at its core, Listing 20-2 does the same thing as the simpler Listing 20-3.

Listing 20-3: The essence of the code in Listing 20-2

```
import javax.swing.JFrame;

class ShowEmptyFrame {

  public static void main(String[] args) {
    JFrame frame = new JFrame();

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setBounds(100, 100, 450, 300);
    frame.setVisible(true);
  }
}
```

Like its intricate predecessor, Listing 20-3 displays the empty frame shown in Figure 20-7. To help display the frame, Listing 20-3 calls the frame's `setDefaultCloseOperation` and `setVisible` methods.

The only brand-new feature in Listing 20-3 is the call to the frame's `setBounds` method. Calling `setBounds(100, 100, 450, 300)` creates a frame with the following measurements:

- The frame's upper-left corner is 100 pixels from the left edge of the screen.
- The frame's upper-left corner is 100 pixels from the top of the screen.
- The frame is 450 pixels wide.
- The frame is 300 pixels high.

Some details about the code

You can use WindowBuilder without understanding any of the subtleties in Listing 20-2. But if full comprehension is your goal for Listing 20-2, you can start by reading this section.

In Listing 17-3 in Chapter 17, I create an instance of the `Purchase` class with the line

```
Purchase onePurchase = new Purchase();
```

The code in Listing 20-2 does the same kind of thing. In Listing 20-2, WindowBuilder creates an instance of the `MyFrame` class with the following line:

```
MyFrame frame = new MyFrame();
```

Compare Figure 17-4 in Chapter 17 with this chapter's Figure 20-8.

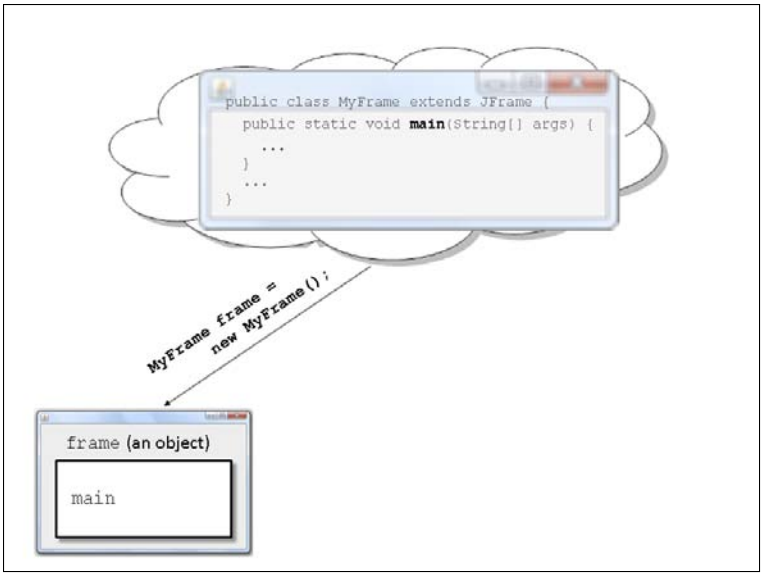


Figure 20-8: An object created from the MyFrame class.

In both figures, a `new SomethingOrOther()` constructor call creates an object from an existing class.

- **In Chapter 17, I create an instance of my Purchase class.**

  This object represents an actual purchase (with a purchase amount, a tax, and so on).

- **In this chapter, I create an instance of the MyFrame class (the class in Listing 20-2).**

  This object represents a frame on the computer screen (a frame with borders, a minimize button, and so on). In a more complicated application — an app that displays several frames — the code might create several objects from a class such as *MyFrame* (see Figure 20-9).

Extending a class

In Listing 20-2, the words `extends JFrame` are particularly important. When you see Java's `extends` keyword, imagine replacing that keyword with the phrase "is a kind of."

```
public class MyFrame is a kind of JFrame {
```

When you type `MyFrame extends JFrame`, you declare that your new `MyFrame` class has all the methods and other things that are built into Java's own `JFrame` class, and possibly more. For example, a `JFrame` instance has `setDefaultCloseOperation`, `setBounds` and `setVisible` methods, so every new `MyFrame` instance has `setDefaultCloseOperation`, `setBounds` and `setVisible` methods (see Figure 20-10).

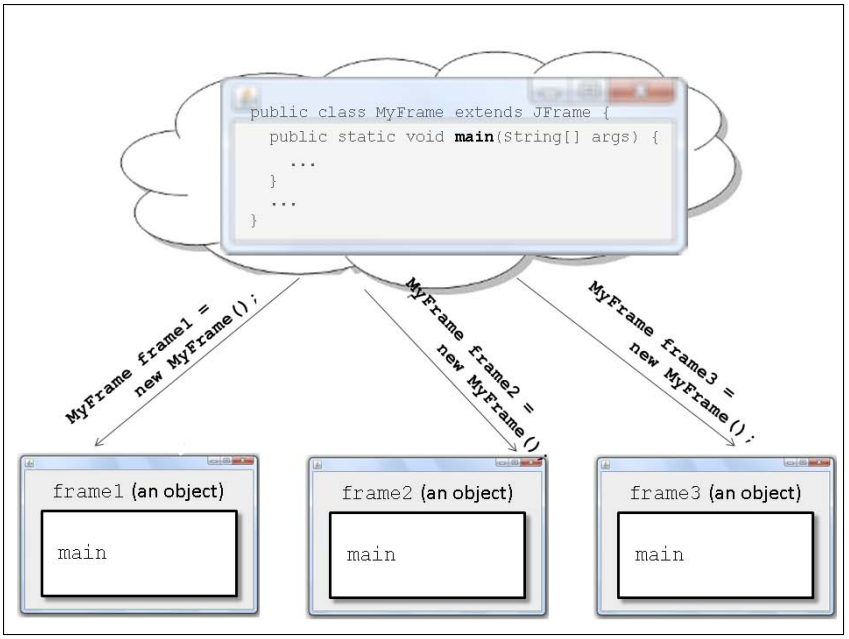


Figure 20-9: Creating three objects from the MyFrame class.

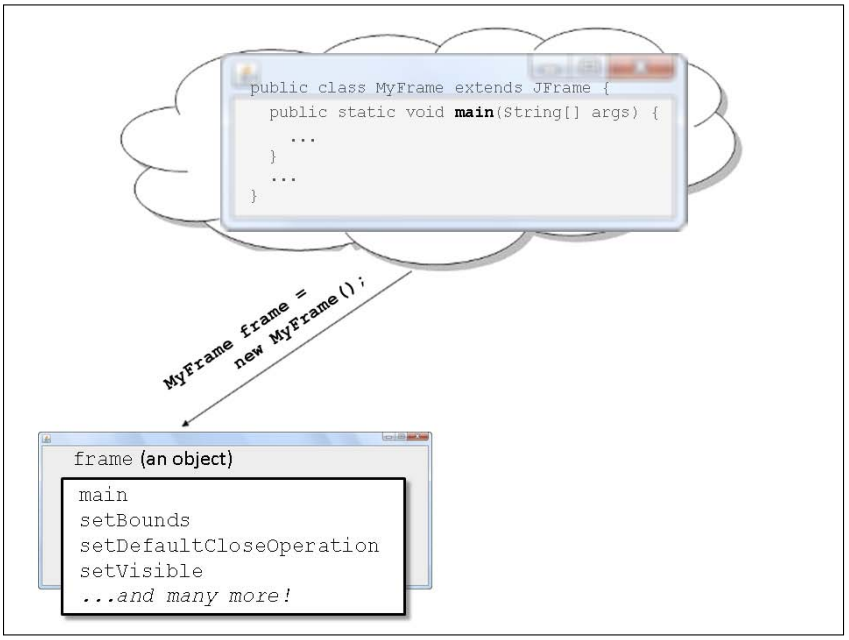


Figure 20-10: A MyFrame instance has many methods.

And the best part is, you get all of the `JFrame` methods for free. The `MyFrame` class's code doesn't need declarations for methods, such as `setDefaultCloseOperation`, `setBounds`, and `setVisible` (see Listing 20-2). The only declarations in the MyFrame class's code are for brand-new methods — methods that aren't already declared in the `JFrame` class's code. It's as if Listing 20-2 contained the following information:

```
public class MyFrame is a kind of JFrame {

    And in addition to what's in JFrame, MyFrame also has
    public static void main(String[] args) {
        // Etc.
    }

    And in addition what's in JFrame, MyFrame also has
    public MyFrame() {
        // Etc.
    }
}
```

The `extends` keyword adds a very important idea to Java programming — the notion of *inheritance*. In Listing 20-2, the newly created `MyFrame` class *inherits* the methods (and other things) that are declared in the existing `JFrame` class. Inheritance is a pivotal feature of an object-oriented programming language.

Java class, instantiate thyself!

The example in Chapter 17 consists of two Java classes — one class that defines what it means to be a Purchase (Listing 17-2), and another class that creates an object from the Purchase class (Listing 17-3). The second class uses its object to calculate things about a purchase. In other words, Listing 17-2 says "this is what a purchase looks like," and Listing 17-3 says "create a purchase and the do these things with the purchase."

But the code in Listing 20-2 has an interesting quirk. The code plays two roles. First, the code defines what it means to be a MyFrame:

```
public class MyFrame extends JFrame {
```

The code also creates an object from the MyFrame class:

```
public static void main(String[] args) {

    MyFrame frame = new MyFrame();
```

I illustrate the situation in Figure 20-11.

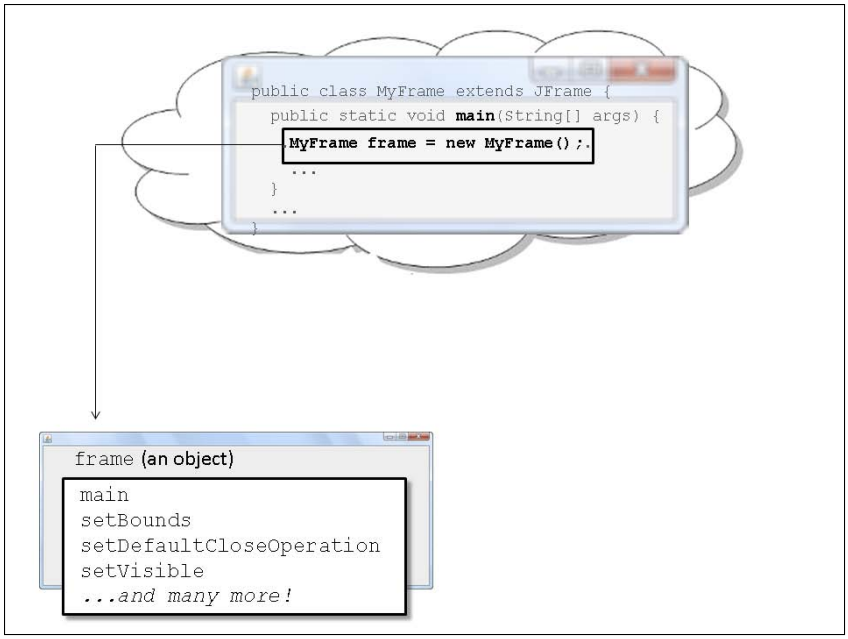


Figure 20-11: The MyFrame class creates a MyFrame object.

The code in Listing 20-2 creates an instance of itself. As strange as it seems, this self-replicating behavior is a common Java programming idiom. You take a class that defines something useful (such as a MyFrame), and then you plunk a main method inside the class's code.

For me, the whole business is weird. You wouldn't create a class that describes both checking accounts and race horses, so why does WindowBuilder create a class that describes both a frame and a program's main flow of control? My only guess is that Java programmers think of main methods as oddballs in object-oriented programming. For programmers, a main method is a necessary evil — the single step that begins a thousand-mile journey.

How to avoid traffic jams

If Marcel Proust had been a computer programmer, he would have written the main method in Listing 20-2. This method is a study in the use of braces, parentheses, and semicolons. To get the main method in Listing 20-2, you start with two innocent statements:

```
MyFrame frame = new MyFrame();
frame.setVisible(true);
```

You put those statements inside a `try-catch` statement (whatever that is). The `try-catch` statement is inside a method named `run`, which is inside a `new Runnable()` constructor call, which is in turn inside an `EventQueue.invokeLater` method call. Whew!

I don't dare attempt to describe the main method's code in detail. (John Wiley & Sons, Inc. would have to kill too many trees.) Instead, I describe the reason behind the code — what's deficient in this chapter's odd-numbered listings and (roughly) why Listing 20-2 is better.

A Java program is *multithreaded*, which means that during a Java program's run, several things happen at once. While part of your code is calculating rent payments, another part of your code draws a frame on the screen, and yet another part of Java's virtual machine listens for the user's key presses and mouse clicks.

With all that stuff happening at once, you might experience bottlenecks. You might even experience *deadlocks* — situations in which two parts of your code play a virtual game of chicken with one another. "I won't run until you run," says one part of your code to another. "Well, then we're stuck, because I won't run until you run," says the other part to the first part. Your Java program freezes, and your program's window is completely unresponsive.

To avoid the possibility of such a standoff, WindowBuilder creates the main method in Listing 20-2. This main method carefully funnels all frame-related business to one *event-dispatching thread*. Think of the event-dispatching thread as a traffic cop. One part of your code says, "I want to maximize the frame." Another part of your code says, "I want to change the frame's color." With the main method in Listing 20-2, one part asks the event-dispatching thread to "maximize the frame when you have the opportunity to do so." And the other part asks the same event-dispatching thread to "change the frame's color when you can." The event-dispatching thread controls the workflow and keeps traffic moving along.

Now there's comforting news. The main method in Listing 20-2 is boilerplate code. If your GUI application isn't too fancy, you can simply copy this main method into your application's `.java` file. (Remember to change MyFrame to whatever name you've given to your Java class.) And if you create your program using WindowBuilder, the WindowBuilder tool writes the main method for you. No additional tinkering is required.

Adding Stuff to Your Frame

I like empty spaces. When I lived on my own right out of college, my apartment had no pictures on the walls. I didn't want to stare at the same works of art day after day. I preferred to fill in the plain white spaces with images from my own imagination. So for me, the empty frame in Figure 20-7 is soothing.

But if Figure 20-7 isn't acquired by New York's Museum of Modern Art, the frame is quite useless. (By the way, I'm still waiting to hear back from the museum's curator.) When you create a high-powered GUI program, you start

by creating a frame with buttons and other widgets. Then you add methods to respond to keystrokes, button clicks, and other such things.

The next section contains some code to respond to a user's button clicks. But in this section, you use WindowBuilder to display a button and a text field:

1. **Follow the instructions in this chapter's earlier "Creating a GUI class" section.**
2. **With MyFrame.java in the editor, look for two tabs at the bottom of the editor area.**

   The labels on the tabs are Source and Design. The Source tab displays Java code (the stuff in Listing 20-2).
3. **Select the Design tab.**

   The Design tab displays a preview of the frame and a palette full of things that you can add to the frame (see Figure 20-12).

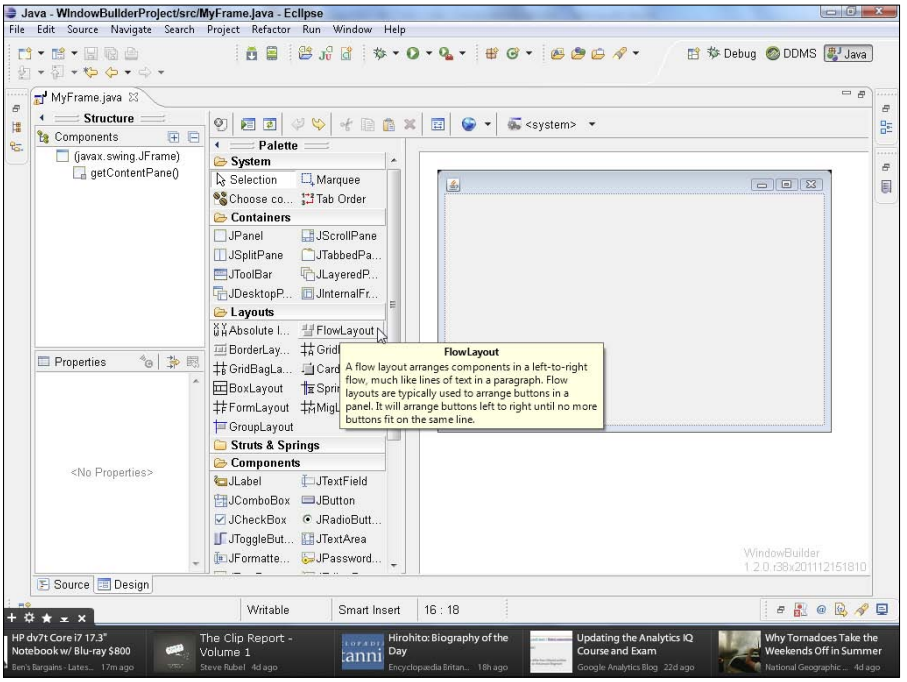


Figure 20-12: The Design tab.

You can quickly enlarge the Design area by double-clicking the MyFrame.java tab at the top of the editor.

4. **In the Layouts section of the Palette, click the FlowLayout item.**

   In response, the FlowLayout item appears to be pressed down (see Figure 20-13).

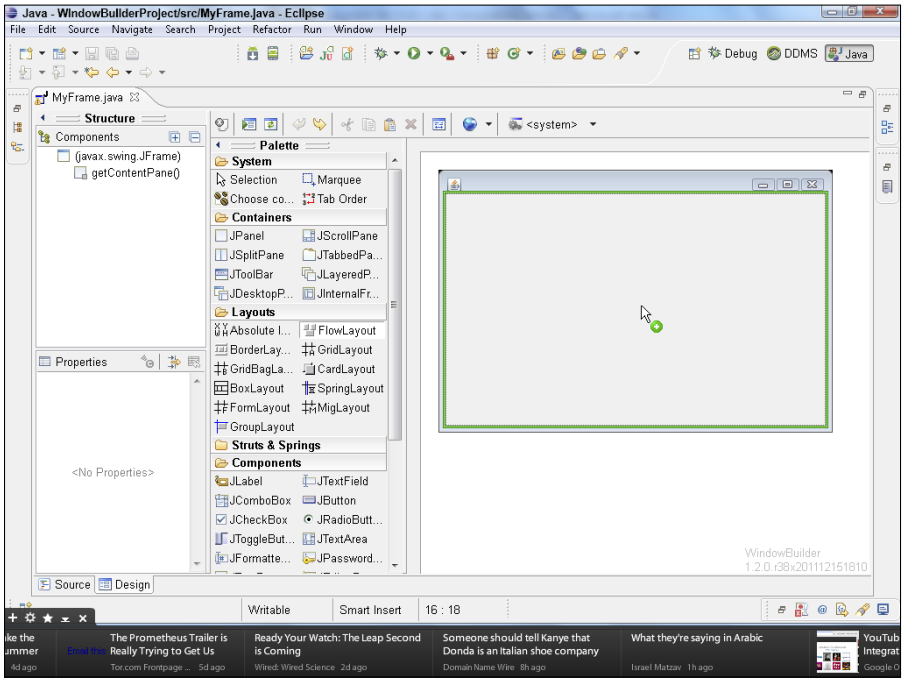


Figure 20-13: Preparing to add a FlowLayout to your frame.

A *FlowLayout* arranges objects in a row, one right after another across the frame. After filling an entire row with objects, Java places additional objects on a second row (or a third, fourth, or fifth row, if necessary).

By default, your frame has a *BorderLayout*. In a BorderLayout, items such as text fields and buttons land in one of five regions — the NORTH, SOUTH, EAST, WEST, and CENTER regions. But you can apply any item in the palette's Layouts section to your frame. For example, with an *AbsoluteLayout* (also known as a *null layout*), you can drag buttons and other components to any location on the frame. With a *GridLayout*, you put things into cells in a table. To learn more, put any of the palette's layouts on your frame and watch what happens.

5. **Hover your mouse pointer over the preview of your frame.**

   Again, see Figure 20-13.

6. **With your mouse pointer still hovering over the frame's preview, click the mouse.**

   At this point, WindowBuilder's response is a big letdown. The Design tab returns to its original look with nothing special in the frame and with none of the palette's items pressed down.

   To change your mind between Steps 5 and 6 (deciding against adding a FlowLayout to the frame), press Escape.

   As you'd do when following any long sequence of instructions, pause frequently to save your work in progress. In Eclipse's main menu, choose File➪Save.

7. **Look for two items in the Components section of the Palette — the JTextField item and the JButton item (see Figure 20-13).**

   Java's JTextField class describes a white box (commonly known as a *text field*) in which the user types a line of text. The JButton class describes one of those clickable things that we all know and love. The frame in Figure 20-14 contains an empty JTextField component, and a JButton component with the words *New button* on its face.

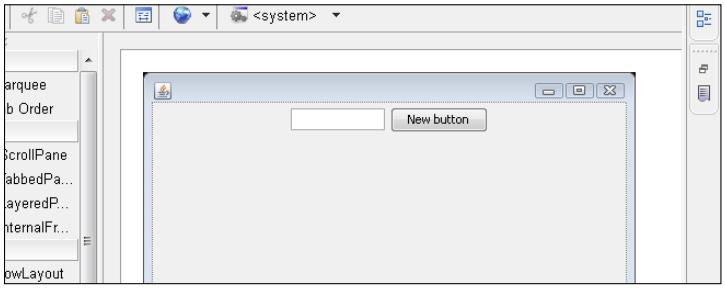


Figure 20-14: Your frame contains a text field and a button.

You can make your frame look like the frame in Figure 20-14:

8. **Repeat Steps 4 through 6, this time adding a JTextField to your frame.**
9. **Repeat Steps 4 through 6, this time adding a JButton to your frame.**

   The resulting preview frame is in Figure 20-14.

   Having added some widgets to the frame, you can make some refinements.

10. **With your mouse, select the JTextField in the preview of the frame (see Figure 20-15).**

    You can also select a component by clicking the component's name in the Components pane. You find the Components pane in the upper-right corner of the editor area.

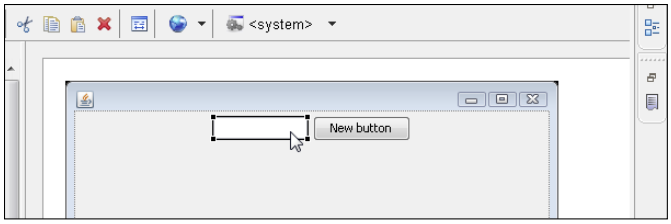


Figure 20-15: Selecting the JTextField component.

11. **Look for the Properties pane in lower-left corner of the editor area.**

    The Properties pane displays characteristics of the currently selected component (see Figure 20-16).

12. **Type some stuff in the Properties pane's *text* row (see Figure 20-17).**

13. **Click your mouse anywhere outside of the Property pane's text row (preferably at some neutral point inside the Editor area).**

    Now the JTextField component in the frame's preview contains your text (see Figure 20-18). But as you can see, the JTextField component may not be wide enough.

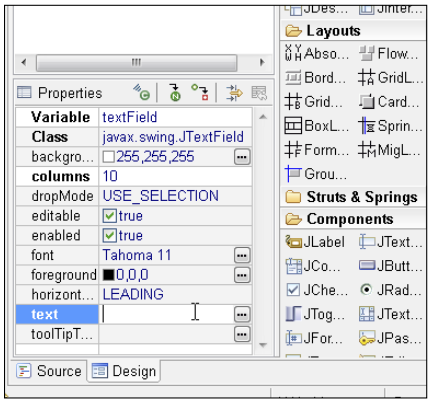


Figure 20-16: The Properties pane displays the JTextField component's properties.

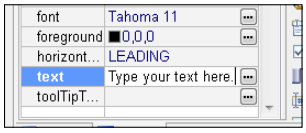


Figure 20-17: Typing text into one of the Properties pane's rows.

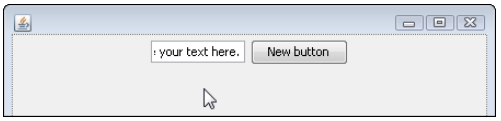


Figure 20-18: Text in the text field.

14. **Again, select the JTextField component in the preview of the frame and then increase the number in the Columns row of the Properties pane.**

    The width of the JTextField component increases accordingly.

    A few more refinements are in order:

15. **Select the JButton component and use the Properties pane to change the component's text.**

    In Figure 20-19, I change the text to the word `Capitalize`.

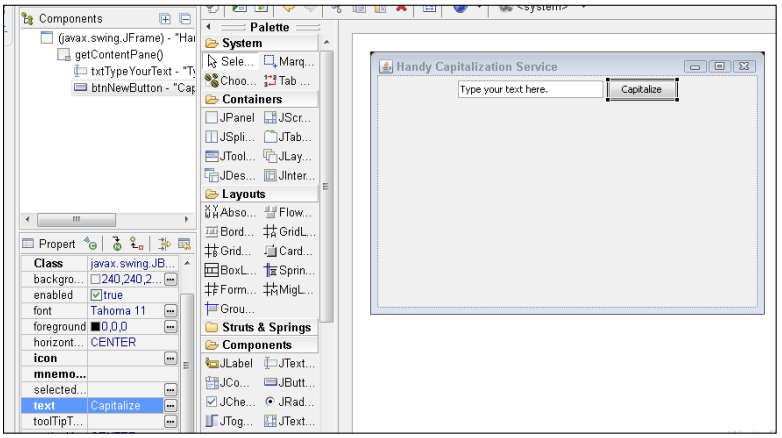


Figure 20-19: Changing the text on the face of the button.

16. **Select the entire frame by clicking somewhere on the frame's border and then use the Properties pane to change the frame's Title property.**

    In Figure 20-19, the frame's title is *Handy Capitalization Service*.

As you follow this section's steps, WindowBuilder modifies your project's Java code. Having followed this section's steps, you can run the project in the usual way (by choosing Run➪Run As➪Java Application). But when the project runs, the frame doesn't do anything. When you click the button on the frame, nothing happens. When you type in the text field, nothing happens. What a waste!

In the next section, you get the button to do something.

Taking Action

The program that you create in this chapter is approximately 50 lines long. But up to this point in the chapter, you don't type a single line of code! That's about to change because, in this section's instructions, you make a button respond to a mouse click. And, yes, you do this by typing a single line of code!

You can write Android apps (for mobile phones and tablet devices) in Java, and using the App Inventor tool you can write some complete Android programs without writing any code at all (not even a single line). I happen to know about an excellent book in which 2 out of 24 chapters cover App Inventor. It's *Android Application Development All-in-One For Dummies* (Wiley) by Barry Burd. (Oh, no! Yet another shameless plug!)

1. **Follow the instructions in this chapter's earlier "Adding Stuff to Your Frame" section.**

   As a result, you have a frame containing a text field (a JTextField component) and a button (a JButton component). You're using the Design tab — a tool belonging to Eclipse's WindowBuilder.

2. **In the preview of the frame (or in the Components pane), right-click the JButton component; as usual, Mac users Control-click the JButton component.**

3. **In the resulting context menu, choose Add Event Handler.**

   I'm proud that I figured out how to create Figure 20-20. So please have a look at it.

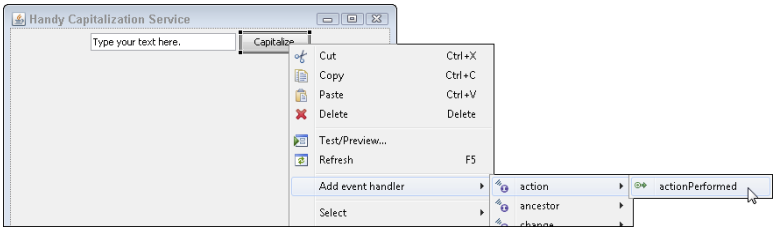
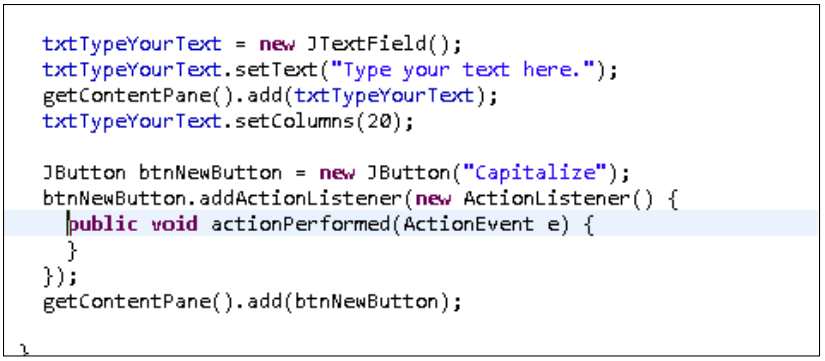


Figure 20-20: Cascading context menus.

4. **In the resulting context submenu, choose action.**

5. **In the resulting context sub-submenu, choose actionPerformed (see Figure 20-20).**

   When you click actionPerformed, Eclipse takes you to the Java code that WindowBuilder has created (see Figure 20-21).



```
txtTypeYourText = new JTextField();
txtTypeYourText.setText("Type your text here.");
getContentPane().add(txtTypeYourText);
txtTypeYourText.setColumns(20);

JButton btnNewButton = new JButton("Capitalize");
btnNewButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
  }
});
getContentPane().add(btnNewButton);
}
```

Figure 20-21: The code that WindowBuilder built.

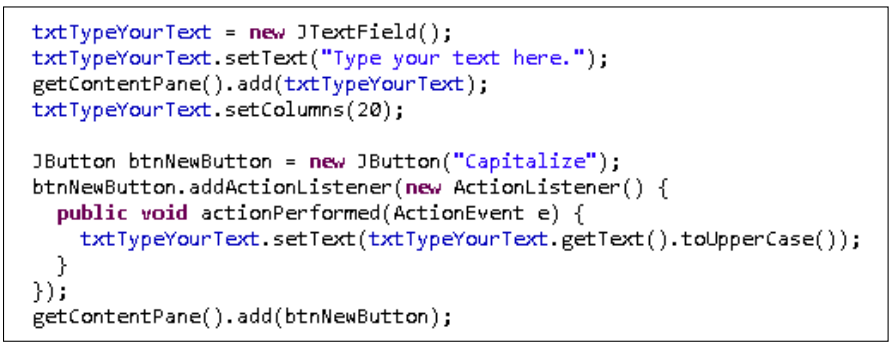
6. **In the Java code, look for the name of the JTextField component.**

   In Figure 20-21, the component's name is txtTypeYourText. WindowBuilder creates a name for each of your components. By default, your first empty text field is named textField. Your second empty text field is named textField_*1*. A text field containing the text *Hello! How are you?* is named txtHelloHowAre. I don't know all the naming rules, but the ones that I know seem to make sense.

7. **Type the following line of code inside the actionPerformed method:**

   ```
   nameOfYourJTextComponent.setText
     (nameOfYourJTextComponent.getText().toUpperCase());
   ```

   For a text field whose name is txtTypeYourText, the added line of code is shown in Figure 20-22.



```
txtTypeYourText = new JTextField();
txtTypeYourText.setText("Type your text here.");
getContentPane().add(txtTypeYourText);
txtTypeYourText.setColumns(20);

JButton btnNewButton = new JButton("Capitalize");
btnNewButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    txtTypeYourText.setText(txtTypeYourText.getText().toUpperCase());
  }
});
getContentPane().add(btnNewButton);
```

Figure 20-22: Adding code to the actionPerformed method.

8. **Save and run your program!**

When you run this section's program, you see something like the screen shots in Figures 20-23, 20-24, and 20-25. *Et voilà!* When you click the button, Java capitalizes your text!

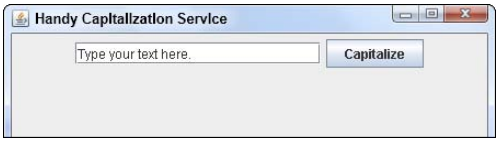


Figure 20-23: A brand-new frame.

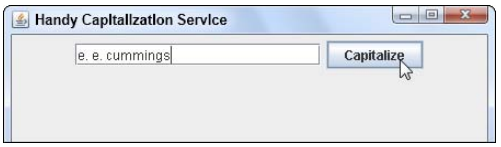


Figure 20-24: The user types in the text box.

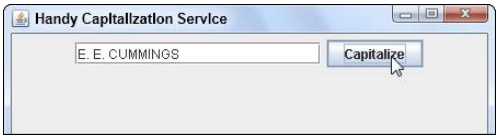


Figure 20-25: Clicking the button capitalizes the text in the text box.

For your reading pleasure, I've pasted all the code for this section's example into Listing 20-4. The only statement that you type yourself is in bold near the end of the listing.

Listing 20-4: Capitalism in Action

```
import java.awt.EventQueue;

import javax.swing.JFrame;
import java.awt.FlowLayout;
import javax.swing.JTextField;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;


public class MyFrame extends JFrame {
  private JTextField txtTypeYourText;
```

(continued)

Listing 20-4 ***(continued)***

```
  /**
   * Launch the application.
   */
  public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
      public void run() {
        try {
          MyFrame frame = new MyFrame();
          frame.setVisible(true);
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    });
  }

  /**
   * Create the frame.
   */
  public MyFrame() {
    setTitle("Handy Capitalization Service");
    setBounds(100, 100, 450, 300);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    getContentPane().setLayout(new FlowLayout(FlowLayout.
          CENTER, 5, 5));

    txtTypeYourText = new JTextField();
    txtTypeYourText.setText("Type your text here.");
    getContentPane().add(txtTypeYourText);
    txtTypeYourText.setColumns(20);

    JButton btnNewButton = new JButton("Capitalize");
    btnNewButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        txtTypeYourText.setText
          (txtTypeYourText.getText().toUpperCase());
      }
    });
    getContentPane().add(btnNewButton);

  }

}
```

Want to read more? I have a whole chapter about it in *Java For Dummies,* 5th Edition (written by yours truly and published by John Wiley & Sons, Inc.).

Part V
The Part of Tens

"Evidently he died of natural causes following a marathon session animating everything on his personal website. And no, Morganstern — the irony isn't lost on me."

In this part . . .

You're near the end of the book, and it's time to sum it all up. This part of the book is your slam-bam two-thousand-words-or-less resource for Java. What? You didn't read every word in the chapters before this one? That's okay. You can pick up a lot of useful information in this Part of Tens.

Chapter 21

Ten Sets of Web Links

In This Chapter

- Finding resources from Oracle
- Getting sample code
- Reading the latest Java news
- Moving up — jobs, certification, and more
- Finding out about other useful technologies and languages

This chapter has ten bundles of resources. Each bundle has websites for you to visit. Each website has resources to help you write programs more effectively.

The Horse's Mouth

Oracle's official website for Java is `www.oracle.com/technetwork/java`. This site has all the latest software for download. The site also has a great section with online tutorials and mini-courses. The tutorial/mini-course section's web address is

```
www.oracle.com/technetwork/java/index-jsp-135888.html
```

In addition, Oracle has two special-purpose Java websites. Consumers of Java technology should visit `www.java.com`. Programmers and developers interested in sharing Java technology can go to `www.java.net`.

Finding News, Reviews, and Sample Code

The web has plenty of sites devoted exclusively to Java. Many of these sites feature reviews, links to other sites, and best of all, gobs of sample Java code. They may also offer free mailing lists that keep you informed of the latest Java developments. Here's a brief list of such sites:

- **The JavaRanch:** `www.javaranch.com`
- **Developer.com/Gamelan:** `www.developer.com/java`
- **The Giant Java Tree:** `www.gjt.org`
- **The Java Boutique:** `http://javaboutique.internet.com`
- **FreewareJava.com:** `www.freewarejava.com`
- **Java Shareware:** `www.javashareware.com`

Improving Your Code with Tutorials

To find out more about Java, you can visit Oracle's online training pages. Some other nice sets of tutorials are available at the following websites:

- **Richard Baldwin's website:** `http://www.dickbaldwin.com`
- **Java Beginner Tutorial:** `http://www.javabeginner.com`
- **IBM developerWorks:** `www.ibm.com/developerworks/training`

Finding Help on Newsgroups

Have a roadblock you just can't get past? Try posting your question on an Internet newsgroup. Almost always, some friendly expert will post just the right reply.

With or without Java, you should definitely start exploring newsgroups. You can find thousands of newsgroups — groups on just about every conceivable topic. (Yes, there are more newsgroups than *For Dummies* titles!) To get started with newsgroups, visit `http://groups.google.com`. For postings specific to Java, look for the groups whose names begin with `comp.lang.java`. As a novice, you'll probably find the following two groups to be the most useful:

- `http://comp.lang.java.programmer`
- `http://comp.lang.java.help`

Reading Documentation with Commentary

When programmers write documentation, they ask themselves questions and then answer those questions as best they can. But sometimes, they don't ask themselves all the important questions. And often, they assume that the reader already knows certain things. If you're a reader who doesn't already know these things, you may be plain out of luck.

One way or another, all documentation omits some details. That's why other peoples' comments about the documentation can be so helpful. At `www.jdocs.com`, experienced Java programmers annotate existing Java documentation with their own comments. The comments include tips and tricks, but they also add useful pieces of information — pieces that the documentation's original authors omitted. If you need help with an aspect of the Java API, this is a great website to visit.

Listen!

You can visit `http://javaposse.com` for all kinds of useful, up-to-the-minute information, but you can also listen to the Java Posse's weekly podcasts. Since 2005, four Java gents have been podcasting news, commentary, and thoughtful discussion to Java professionals all over the world. In the tech community, The Java Posse is a venerable institution — a great source of facts and musings about Java-related issues.

Opinions and Advocacy

Java isn't just techie stuff. The field has issues and opinions of all shapes and sizes. To find out more about them, visit any of these sites:

- **Oracle Blogs:** `http://blogs.oracle.com`
- **JRoller:** `http://jroller.com`
- **The java.blogs community:** `www.javablogs.com`
- **The DZone at Javalobby:** `http://java.dzone.com`
- **The list of blogs at No Fluff Just Stuff:** `www.nofluffjuststuff.com/blog/list`

By the way, when you become proficient in Java, I highly recommend attending one of the No Fluff Just Stuff conferences. Each conference features advanced presentations by seasoned industry experts.

Looking for Java Jobs

Are you looking for work? Would you like to have an exciting, lucrative career as a computer programmer? If so, check the SkillMarket at `http://mshiltonj.com/sm`. This site has statistics on the demand for various technology areas. The site compares languages, databases, certifications, and more. Best of all, the site is updated every day.

For another take on the popularity of various programming languages, visit the TIOBE Programming Community Index at `www.tiobe.com/index.php/content/paperinfo/tpci`. The index's monthly list rates the amount of buzz surrounding each language. Of course, as I check the site for inclusion in this chapter, Java is right on top.

After you've checked all the SkillMarket and Tiobe numbers, try visiting a website designed especially for computer job seekers. Point your web browser to `http://java.computerwork.com`.

Finding Out More about Other Programming Languages

It's always good to widen your view. So to find out more about some languages other than Java, visit the Éric Lévénez site: `www.levenez.com/lang`. This site includes a cool chart that traces the genealogy of the world's most popular programming languages. For other language lists, visit the following websites:

- **The Language List:** `http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm`
- **The Quine Page**: `www.nyx.net/~gthompso/quine.htm`
- **Steinar Knutsen's Language list page:** `http://home.nvg.org/~sk/lang/lang.html`

Finally, for quick information about anything related to computing, visit `http://foldoc.org` — the Free On-Line Dictionary of Computing.

Everyone's Favorite Sites

It's true — these two sites aren't devoted exclusively to Java. However, no geek-worthy list of resources would be complete without Slashdot and SourceForge.

Slashdot's slogan, "News for nerds, stuff that matters," says it all. At `http://slashdot.org`, you find news, reviews, and commentary on almost anything related to computing. There's even a new word to describe a website that's reviewed or discussed on the Slashdot site. When a site becomes overwhelmed with hits from Slashdot referrals, one says that the site has been *slashdotted*.

Although it's not quite as high-profile, `http://sourceforge.net` is the place to look for open source software of any kind. The SourceForge repository contains more than 80,000 projects. At the SourceForge site, you can download software, read about works in process, contribute to existing projects, and even start a project of your own. SourceForge is a great site for programmers and developers at all levels of experience.

Chapter 22

Ten Useful Classes in the Java API

In This Chapter

- Finding out more about classes
- Discovering other helpful classes

I'm proud of myself. I've written around 400 pages about Java using fewer than 30 classes from the Java API. The standard API has about 4,000 classes, so I think I'm doing very well.

Anyway, to help acquaint you with some of my favorite Java API classes, this chapter contains a brief list. Some of the classes in this list appear in examples throughout this book. Others are so darn useful that I can't finish the book without including them.

For more information on the classes in this chapter, check Java's online API documentation at `http://docs.oracle.com/javase/7/docs/api`.

Applet

What Java book is complete without some mention of applets? An *applet* is a piece of code that runs inside a web browser window. For example, a small currency calculator running in a little rectangle on your web page can be a piece of code written in Java.

At one time, Java applets were really hot stuff, but nowadays, people are much more interested in using Java for business processing. Anyway, if applets are your thing, then don't be shy. Check the `Applet` page of Java's API documentation.

ArrayList

Chapter 16 introduces arrays. This is good stuff, but in any programming language, arrays have their limitations. For example, take an array of size 100. If you suddenly need to store a 101st value, then you're plain out of luck. You can't change an array's size without rewriting some code. Inserting a value into an array is another problem. To squeeze `"Tim"` alphabetically between `"Thom"` and `"Tom"`, you may have to make room by moving thousands of `"Tyler"`, `"Uriah"`, and `"Victor"` names.

But Java has an `ArrayList` class. An `ArrayList` is like an array, except that `ArrayList` objects grow and shrink as needed. You can also insert new values without pain using the `ArrayList` class's `add` method. `ArrayList` objects are very useful because they do all kinds of nice things that arrays can't do.

File

Talk about your useful Java classes! The `File` class does a bunch of things that aren't included in this book's examples. Method `canRead` tells you whether you can read from a file or not. Method `canWrite` tells you if you can write to a file. Calling method `setReadOnly` ensures that you can't accidentally write to a file. Method `deleteOnExit` erases a file, but not until your program stops running. Method `exists` checks to see whether you have a particular file. Methods `isHidden`, `lastModified`, and `length` give you even more information about a file. You can even create a new directory by calling the `mkdir` method. Face it, this `File` class is powerful stuff!

Integer

Chapter 18 describes the `Integer` class and its `parseInt` method. The `Integer` class has lots of other features that come in handy when you work with `int` values. For example, `Integer.MAX_VALUE` stands for the number `2147483647`. That's the largest value that an `int` variable can store. (Refer to Chapter 7.) The expression `Integer.MIN_VALUE` stands for the number `-2147483648` (the smallest value that an `int` variable can store). A call to `Integer.toBinaryString` takes an `int` and returns its base-2 (binary) representation. And what `Integer.toBinaryString` does for base 2, `Integer.toHexString` does for base 16 (hexadecimal).

Math

Do you have any numbers to crunch? Do you use your computer to do exotic calculations? If so, try Java's `Math` class. (It's a piece of code, not a place to sit down and listen to lectures about algebra.) The `Math` class deals with Π, *e*, logarithms, trig functions, square roots, and all those other mathematical things that give most people the creeps.

NumberFormat

Chapter 18 has a section about the `NumberFormat.getCurrency Instance` method. With this method, you can turn `20.338500000000003` into `$20.34`. If the United States isn't your home, or if your company sells products worldwide, you can enhance your currency instance with a Java `Locale`. For example, with `euro = NumberFormat. getCurrencyInstance(Locale.FRANCE)`, a call to `euro.format(3)` returns `3,00` instead of `$3.00`.

The `NumberFormat` class also has methods for displaying things that aren't currency amounts. For example, you can display a number with or without commas, with or without leading zeros, and with as many digits beyond the decimal point as you care to include.

Scanner

Java's `Scanner` class can do more than what it does in this book's examples. Like the `NumberFormat` class, the `Scanner` can handle numbers from various locales. For example, to input `3,5` and have it mean "three and half," you can type `myScanner.useLocale(Locale.FRANCE)`. You can also tell a `Scanner` to skip certain input strings or use numeric bases other than 10. All in all, the `Scanner` class is very versatile.

String

Chapter 18 examines Java's `String` class. The chapter describes (in gory detail) a method named `equals`. The `String` class has many other useful methods. For example, with the `length` method, you find the number of characters in a string. With `replaceAll`, you can easily change the phrase `"my fault"` to `"your fault"` wherever `"my fault"` appears inside a string. And with `compareTo`, you can sort strings alphabetically.

StringTokenizer

I often need to chop strings into pieces. For example, I have a `fullName` variable that stores my narcissistic `"Barry A. Burd"` string. From this `fullName` value, I need to create `firstName`, `middleInitial`, and `lastName` values. I have one big string (`"Barry A. Burd"`), and I need three little strings — `"Barry"`, `"A."`, and `"Burd"`.

Fortunately, the `StringTokenizer` class does this kind of grunt work. Using this class, you can separate `"Barry A. Burd"` or `"Barry,A.,Burd"` or even `"Barry<tab>A.<tab>Burd"` into pieces. You can also treat each separator as valuable data, or you can ignore each separator as if it were trash. To do lots of interesting processing using strings, check out Java's `StringTokenizer` class.

System

You're probably familiar with `System.in` and `System.out`. But what about `System.getProperty`? The `getProperty` method reveals all kinds of information about your computer. Some of the information you can find includes your operating system name, your processor's architecture, your Java virtual machine version, your classpath, your username, and whether your system uses a backslash or a forward slash to separate folder names from one another. Sure, you may already know all this stuff. But does your Java code need to discover it on the fly?

Index

• Symbols and Numerics •

+ (plus sign)
  addition operator, 133, 364, 366
  concatenating strings, 364
_ (underscore)
  in class names, 56
  in literals, 78
. (dot)
  calling methods, 344, 363
  frame height or width, 383
( ) (parentheses), in method names, 363
? : (question mark colon), conditional operator, 222
&& (ampersands), logical and, 188
* (asterisk), multiplication operator, 133
*= (asterisk equal sign), increment operator, 138–141
= (equal sign), comparing values, 160
== (equal signs), equality operator, 160, 162
! (exclamation point), logical not, 188
!= (exclamation point equal sign), not-equal operator, 160, 162
> (greater than), greater than operator, 160
>= (greater than equal), greater than or equal operator, 160
< (less than), less than operator, 160
<= (less than equal), less than or equal operator, 160
-= (minus equal sign), decrement operator, 138–141
- (minus sign), subtraction operator, 133
- - (minus signs), decrement operator, 138–141
% (percent sign), remainder operator, 133
+= (plus equal sign), increment operator, 138–141
++ (plus signs), increment operator, 138–141
/ (slash), division operator, 133
/*...*/ (slash asterisk...), traditional comment, 80
/**...*/ (slash asterisks), Javadoc comments, 80
/= (slash equal sign), decrement operator, 138–141
// (slashes), end-of-line comment, 80
|| (vertical bars), logical or, 188–189
{ } (curly braces), punctuating code
  in classes, 89
  common problems, 107–109
  identifying blocks of code, 79–80
; (semicolon), ending statements, 86
32-bit systems *versus* 64-bit, 26

• A •

abbreviating names. *See* import declarations
AbsoluteLayout, 397
Abstract Windowing Toolkit (AWT), 380
actions, Eclipse IDE, 63
adding numbers, 133, 143
aligning output, 237
allographs, 167
ampersands (&&), logical and, 188
Android Application Development All-in-One For Dummies, 401
API (application programming interface), 12
API documentation
  description, 18
  tips on using, 382
  web links, 408–409

API Specification. *See* API documentation
applets, 413
archive files, 24
areas, Eclipse IDE, 62
`ArrayList` class, 414. *See also* arrays
arrays. *See also* `ArrayList` class
  components, 312
  creating reports from, 304, 305–306, 308, 314–315
  definition, 312
  elements, 312
  indexes, 312
  looping through, 319–320
  overview, 309–313
  sample programs, 304, 305–306, 308, 315–319
  storing values in, 313–314
  values, 312
assigning variable values, 118
assignment statements, 118
asterisk (*), multiplication operator, 133
asterisk equal sign (*=), increment operator, 138–141
author of this book, contact information, 21
AWT (Abstract Windowing Toolkit), 380

• B •

Beckstrom, Bob, 355
bits
  32-bit systems *versus* 64-bit, 26
  definition, 28
  word length, 28–29
blanks, in class names, 56
blocks
  `if` statements, 185
  `while` statements, 230
blogs, web links, 409
bodies, methods, 360
books and publications
  Android Application Development All-in-One For Dummies, 401
  Landscaping For Dummies, 355
  Managing Your Money Online For Dummies, 355
  UNIX For Dummies: Quick Reference, 5th Edition, 355
`boolean` type
  description, 157–159
  range of values, 166
`boolean` variables, `if` statements, 194–195
BorderLayout, 397
bottlenecks, 394–395
`break` statements, 212–213, 217–219
Burd, Barry, 401
buttons
  attaching actions to, 400–404
  displaying, 395–400
`byte` type, range of values, 145
bytecode, 12, 14

• C •

`.cab` files, 24
calling methods, 83–84, 344, 346
capitalization. *See* case
cascading `if` statements, 202–204
case
  comparing characters, 165–166
  upper, converting to, 149–150
`case` clauses, 212–213
case sensitivity
  debugging, 102–103
  definition, 58
  keywords, 75
  methods, 87
casting, 136–137
Celsius, converting to Fahrenheit, 160–163
`char` type. *See also* `String` type
  adding to strings, 366
  assigning to strings, 365
  capacity, 151
  definition, 148
  example, 148
  range of values, 166

characters
allographs, 167
comparing, 165–166
definition, 148
glyphs, 167
reading, 156–157
type, 148
Unicode, 166–167
chevron (double arrow), Eclipse IDE, 66–67
`.class` files, 12
class names
abbreviating, 351
naming conventions, 56
classes. *See also* objects
`ArrayList`, 414. *See also* arrays
combining and using data, 321, 344
creating, 322–326
definition, 88
extending, 391–393
`File`, 414
instantiating, 393–394
`JLabel`, 382
`main` method, multiple, 325
`main` method, single, 323–324
`Math`, 415
`NumberFormat`, 348–350, 415
versus objects, 332–333
without objects, 334
`PrintStream`, 252
`Random`, 179–180
reference types, 323
sample code, 323–324
`StringTokenizer`, 416
Swing, 379–380
`System`, 351
versus tables, 332–333
classes, GUI
bottlenecks, 394–395
creating, 387–388
deadlocks, 394–395
event-dispatching threads, 395
extending, 391–393
instantiating, 393–394
multithreading, 395
running, 388–389
sample code, 389–390
classes, `Integer`
`MAX_VALUE` parameter, 414
`MIN_VALUE` parameter, 414
`parseInt` method, 346–348, 414
`toBinaryString` method, 414
`toHexString` method, 414
classes, `JFrame`
definition, 381
description, 383–384
`DISPOSE_ON_CLOSE` parameter, 384
`DO_NOTHING_ON_CLOSE` parameter, 384
`EXIT_ON_CLOSE` parameter, 383
`HIDE_ON_CLOSE` parameter, 384
`pack` method, 384
`setDefaultCloseOperation` method, 383–384
`setVisible` method, 384
classes, `Scanner`
description, 99, 415
`findWithinHorizon` method, 98
`java.util` package, 350–351
`next` method, 98
`nextDouble` method, 98
`nextInt` method, 98
`nextLine` method, 98
classes, `String`
definition, 77, 335–336
description, 416
reading/writing values, 338–339
sample code, 336–337
variables, 337–338
close button, Eclipse IDE, 66–67
code. *See also* editors; programs
definition, 9
Haskell, example, 11
indenting, 89, 113
as instructions to computers, 10–11
portability, 17–18
programming languages, 11

code *(continued)*
reusing, 17–18
translating, 12–13. *See also* compiling code
Visual Basic, example, 11
writing, issuing commands for, 19. *See also* Eclipse IDE
code, comments
/*...*/ (slash asterisk...), traditional comment, 80
/**...*/ (slash asterisks), Javadoc comment, 80
// (slashes), end-of-line comment, 80
definition, 80
description, 80–81
purpose of, 80
code, compiling
bytecode, 12, 14
`.class` files, 12
compile-time warnings, 61
logic errors, 60
object code, 12
overview, 12–13
source code, 12
unchecked runtime exceptions, 60–61
code, punctuation
{ } (curly braces), 79–80, 89
debugging, 103–109
methods, 363
numbers, 79
spaces, 113
code, running
canned Java programs, 48–52
code you've created, 52–61
description, 13, 15–17
in Eclipse IDE, 48–52
mortgage program, sample, 48–52
text-based programs, 48
comments
/*...*/ (slash asterisk...), traditional comment, 80
/**...*/ (slash asterisks), Javadoc comment, 80
// (slashes), end-of-line comment, 80
definition, 80
description, 80–81
purpose of, 80
comparing
characters, 165–166
decimal numbers, 160–163
numbers, 160–163
strings, 341–344
comparison operators, 160
compile time errors
debugging, 59
indicating, 58, 61
red jagged underlines, 58
yellow jagged underlines, 61
compilers
definition, 12
`javac` file, 13
`javac.exe` file, 13
source code *vs.* object code, 13
on your computer, 13
compiling code
bytecode, 12, 14
`.class` files, 12
compile-time warnings, 61
logic errors, 60
object code, 12
overview, 12–13
source code, 12
unchecked runtime exceptions, 60–61
components of arrays, 312
compressed archive files, 24
computer programs. *See* programs
computers
getting information about, 416
limitations of, 92
role in debugging, 105
concatenating strings, 364
conditional operators, 222. *See also* `switch` statements
conditions, in expressions, 159
conditions, in `if` statements
combining, 189–191, 193–194
grouping, 197–199

converting
  Celsius to Fahrenheit, 160–163
  characters to uppercase, 149–150
  Javadoc comments to web pages, 81
  numbers from strings, 346–348, 414. *See also* `parseInt` method
  numbers to binary, 414
  numbers to hexadecimal, 414
  numbers to strings, 348–349
  types. *See* casting
converting Celsius to Fahrenheit, 160–163
curly braces ({ }), punctuating code
  in classes, 89
  common problems, 107–109
  identifying blocks of code, 79–80
currency, formatting numbers as, 348–349

• D •

data types. *See* types
deadlocks, 394–395
Debug perspective, 163–165
debugging. *See also* troubleshooting
  automated, 163–165
  compile time errors, 59
  in Eclipse, 103
  role of the computer, 105
  unexpected problems, 100–101
debugging, common problems
  capitalization, 103
  case sensitivity, 102–103
  punctuation, 103–109
  runtime error messages, 111–112
  spelling, 103, 109–110
decimal numbers. *See also* `double` type; `float` type
  comparing, 160–163
  definition, 121
  reading from the keyboard, 122–124
  types, 146
decision-making, 171–173. *See also* `if` statements; `switch` statements
declaring methods, 83–84
decrement operators, 138–141
decrementing numbers, 138–141
`default` clause, 213, 216
deleting files with `do` statements, 287–291
Developer.com/Gamelan, web link, 408
disk-oriented programs
  Java disk access code, 252
  running, 246–248
  sample program, 248–249
  templates, reading/writing data, 250–251
  troubleshooting, 255–257
  writing, 257–260
disk-oriented programs, reading from files
  examples, 258–259
  file not found, 252, 255–257
  input files, creating, 253
  Java code for, 252
  template for, 250–251
disk-oriented programs, writing to files
  examples, 259–260, 262–263
  Java code for, 252
  output file, viewing, 254–255
  template for, 250–251
displaying text on the screen, 119. *See also* `print` method; `println` method
`DISPOSE_ON_CLOSE` parameter, 384
dividing numbers, 133, 143
division operator, 133
`do` statements. *See also* loops
  deleting files with, 287–291
  description, 291–292
  overview, 286–287
  syntax, 292
dollars, displaying numbers as, 348–349
`DO_NOTHING_ON_CLOSE` parameter, 384
dot (.)
  calling methods, 344, 363
  frame height or width, 383
`double` keyword, 121
`double` type, range of values, 145
downloading
  Eclipse IDE, 22, 35–37
  JDK (Java Development Kit), 22, 26

downloading *(continued)*
JRE (Java Runtime Environment), 22
sample programs, 22, 24, 43–46
tools, 22–23, 25
The DZone at Javalobby, web link, 409

• E •

echoing keyboard input. *See* `EchoLine` program
`EchoLine` program
code, 93
function, 93–94, 96–97
`EchoLine` program
getting keyboard input, 98–99. *See also* `Scanner` class
running, 94–96
testing, 94–96
typing, 94–96, 100
Eclipse IDE. *See also* Eclipse IDE editor; Eclipse WindowBuilder
actions, 63
areas, 62
automated debugging, 163–165
chevron (double arrow), 66–67
close button, 66–67
configuring Java, 39–43
Debug perspective, 163–165
description, 20, 35
downloading, 22, 35–37
importing sample programs, 43–46
installing, 37–38
JRE Home field, configuring, 42
layout, 67
markers, 66
menu buttons, 66
packages, 55
perspectives, 67–68
plugins, 385
previewing code, 50
projects, 50, 53–60
running the first time, 38–46
tab groups, 65
tabs, 65–66
templates, 251
toolbars, 66
windows, 63
workbench, 62
Eclipse IDE, views
active, 64
bringing to the forefront, 64
definition, 64
stacking, 64
toolbars, 66
Eclipse IDE editor
active, 65
compile-time warnings, 61
description, 64
markers, 66
syntax highlighting, 55
X-like markings, 58
Eclipse IDE editor, compile time errors
debugging, 59
indicating, 58, 61
red jagged underlines, 58
yellow jagged underlines, 61
Eclipse WindowBuilder
description, 385
installing, 385–386
Eclipse WindowBuilder, GUI classes
bottlenecks, 394–395
creating, 387–388
deadlocks, 394–395
event-dispatching threads, 395
extending, 391–393
instantiating, 393–394
multithreading, 395
running, 388–389
sample code, 389–390
editors, 19. *See also* Eclipse IDE editor
EE (Enterprise Edition) *versus,* 27
elements of arrays, 312
`else` clauses, 174, 181–183
end of file
bypassing, 273–274
checking for, 269–271

enhanced `for` statements. *See also* loops
  creating, 293–295
  limitations, 320
  looping through arrays, 319–320
  nesting, 295–299
  overview, 292–293
`enum` type, 205–208
enumeration, 205–208
equal sign (=), comparing values, 160
equal signs (==), equality operator, 160, 162
`equals` method, 346
error messages
  debugging, 111–112
  `if` statements, 176
  runtime, 111–112
errors
  logic, 60
  unchecked runtime exceptions, 60–61
errors, compile time
  debugging, 59
  red jagged underlines, 58
  warnings, 58, 61
  yellow jagged underlines, 61
escape sequences, 237
event-dispatching threads, 395
example programs. *See* sample programs
exclamation point (!), logical not, 188
exclamation point equal sign (!=), not-equal operator, 160, 162
`EXIT_ON_CLOSE` parameter, 383
expressions
  conditions, 159
  statements, 141–142
extending classes, 391–393

• F •

Fahrenheit, converting from Celsius, 160–163
fall-through, 219–221
`false`, reserved word, 75
`File` class, 414
filename extensions
  description, 23, 259
  displaying, 23
`FileNotFoundException`, 252
files. *See also specific files*
  archive, 24
  compressed archive, 24
  deleting with `do` statements, 287–291
  end of file, bypassing, 273–274
  end of file, checking for, 269–271
  naming, 261
  troubleshooting, 255–257
files, reading from
  examples, 258–259
  file not found, 252, 255–257
  input files, creating, 253
  Java code for, 252
  template for, 250–251
files, writing to
  examples, 259–260, 262–263
  Java code for, 252
  output file, viewing, 254–255
  template for, 250–251
`findWithinHorizon` method
  definition, 98
  description, 155–156
`float` type, range of values, 145
flow of control, methods, 362–363
FlowLayout, 397
`for` statements. *See also* loops
  conditions, 307–309
  initializing, 281–283
  nesting, 284–286
  overview, 278–280
  sample code, 278
  structure of, 280–281
  versus `while` statements, 283
frames. *See also* GUI (Graphical User Interface); windows
  AbsoluteLayout, 397
  adding content, 395–400
  BorderLayout, 397
  buttons, attaching actions to, 400–404

frames *(continued)*
buttons, displaying, 395–400
FlowLayout, 397
`JLabel` class, 382
layout, 397
null layout, 397
shrinking to content, 384
frames, `JFrame` class
definition, 381
description, 383–384
`DISPOSE_ON_CLOSE` parameter, 384
`DO_NOTHING_ON_CLOSE` parameter, 384
`EXIT_ON_CLOSE` parameter, 383
`HIDE_ON_CLOSE` parameter, 384
`pack` method, 384
`setDefaultCloseOperation` method, 383–384
`setVisible` method, 384
Free On-Line Dictionary of Computing, web link, 410
FreewareJava.com, web link, 408
fully qualified names, 351

• G •

`getCurrencyInstance` method, 348–349
`getProperty` method, 416
getting input. *See* reading
getting values from methods, 373–377
The Giant Java Tree, web link, 408
Giroux, Phillip, 355
glyphs, 167
greater than (`>`), greater than operator, 160
greater than equal (`>=`), greater than or equal operator, 160
GUI (Graphical User Interface)
AWT (Abstract Windowing Toolkit), 380
definition, 379
`ImageIcon` objects, 381–382
`JLabel` class, 382
serialVersionUID, 380
Swing classes, 379–380
version conflicts, 380
GUI (Graphical User Interface), classes
bottlenecks, 394–395
creating, 387–388
deadlocks, 394–395
event-dispatching threads, 395
extending, 391–393
instantiating, 393–394
multithreading, 395
running, 388–389
sample code, 389–390
GUI (Graphical User Interface), frames
AbsoluteLayout, 397
adding content, 395–400
BorderLayout, 397
buttons, attaching actions to, 400–404
buttons, displaying, 395–400
FlowLayout, 397
layout, 397
null layout, 397
shrinking to content, 384
GUI (Graphical User Interface), `JFrame` class
definition, 381
description, 383–384
`DISPOSE_ON_CLOSE` parameter, 384
`DO_NOTHING_ON_CLOSE` parameter, 384
`EXIT_ON_CLOSE` parameter, 383
`HIDE_ON_CLOSE` parameter, 384
`pack` method, 384
`setDefaultCloseOperation` method, 383–384
`setVisible` method, 384
GUI (Graphical User Interface), windows
close operations, specifying, 383–384
creating. *See* Eclipse WindowBuilder
displaying on a screen, 380–382
visibility, setting, 384
`.gz` files, 24

• H •

Haskell language, example, 11
headers, method
checking for consistent type usage, 377
definition, 83–84

description, 359–360
example, 371–372
Herst, David, 113
HIDE_ON_CLOSE parameter, 384

• I •

IBM developerWorks, web link, 408
icons used in this book, 5–6
identifiers
versus keywords, 112
predetermined, 76–78
user defined, 112–113
IDEs (integrated development environments), 19. *See also* Eclipse IDE
if clauses, 175
if statements. *See also* switch statements; while statements
blocks, 185
boolean variables, 194–195
cascading, 202–204
compound statements, 174
conditions, combining, 189–191, 193–194
conditions, grouping, 197–199
description, 173–177
else clauses, 174, 181–184
enumerating outcomes, 205–208
error messages, 176
if clauses, 174
indenting, 179, 181
initializing variables, 191–192
logical operators, 188–189, 196–197
nesting, 199–202
sample code, 173
sample programs, 177–179, 183–185
versus while statements, 230
ImageIcon objects, 381–382
import declarations, 183–184, 186
increment operators, 138–141
incrementing numbers, 138–141
indenting
code, 89, 113
if statements, 179, 181
nested if statements, 200–201
indexes of arrays, 312
infinite loops, 235–236
initializing variable values, 124–125
input, reading. *See* reading
input, searching, 98, 155–156
installing Java-related software
Eclipse IDE, 37–38
Eclipse WindowBuilder, 385–386
juggling multiple versions, 33–35
online *versus* offline, 27
over existing software, avoiding, 29–33
prioritizing versions, 34–35
starting automatically, 27–28
uninstalling old versions, 33–35
instantiating classes, 393–394
int type
minimum/maximum values, finding, 414
range of values, 145
versus String, 341–342
int type, converting
to binary, 414
from character, 346–348, 414
to hexadecimal, 414
Integer class
MAX_VALUE parameter, 414
MIN_VALUE parameter, 414
parseInt method, 346–348, 414
toBinaryString method, 414
toHexString method, 414
integrated development environments (IDEs), 19. *See also* Eclipse IDE
iterations, 227

• J •

Java Beginner Tutorial, web link, 408
The Java Boutique, web link, 408
Java Development Kit (JDK)
downloading, 22, 26
versus JRE (Java Runtime Environment), 26
Java documentation. *See* API documentation
java file, 16

Java language, portability, 17
Java program elements
  ; (semicolon), ending statements, 86
  classes, 88
  comments, 80–81
  identifiers, 76–78
  instructions to the computer, 85–88
  keywords, 74–76
  literals, 78–79
  methods, 82–88
  punctuation, 79–80
  sample code, 72–73
  statements, 86
  variables, 115–119
Java programming toolset. *See* tools
Java Runtime Environment (JRE)
  configuring in Eclipse IDE, 42
  downloading, 22, 26
  Home field, configuring, 42
  versus JDK (Java Development Kit), 26
Java Shareware, web link, 408
Java virtual machine (JVM)
  definition, 16
  on your hard drive, checking for, 20
The java.blogs community, web link, 409
`javac` file, 13
`javac.exe` file, 13
Javadoc comments, converting to web pages, 81
Javadocs. *See* API documentation
`java.exe` file, 16
`java.lang` package, 351
The JavaRanch, web link, 408
`java.util` package, 350–351
JDK (Java Development Kit)
  downloading, 22, 26
  versus JRE (Java Runtime Environment), 26
`JFrame` class
  definition, 381
  description, 383–384
  `DISPOSE_ON_CLOSE` parameter, 384
  `DO_NOTHING_ON_CLOSE` parameter, 384
  `EXIT_ON_CLOSE` parameter, 383
  `HIDE_ON_CLOSE` parameter, 384
  `pack` method, 384
  `setDefaultCloseOperation` method, 383–384
  `setVisible` method, 384
`JLabel` class, 382
job opportunities, web links, 409–410
JRE (Java Runtime Environment)
  configuring in Eclipse IDE, 42
  downloading, 22, 26
  Home field, configuring, 42
  versus JDK (Java Development Kit), 26
JRoller, web link, 409
JVM (Java virtual machine)
  definition, 16
  on your hard drive, checking for, 20

• K •

keyboard input. *See* reading, from the keyboard
keyboard input, echoing. *See* `EchoLine` program
keywords
  case sensitivity, 75
  definition, 74
  versus identifiers, 112
  list of, 75
  predetermined meanings, 75–76
  reserved words, 75

• L •

Landscaping For Dummies, 355
The Language List, web link, 410
layout, Eclipse IDE, 67
layout, frames, 397
less than (<), less than operator, 160
less than equal (<=), less than or equal operator, 160
Levine, John R., 355

libraries of reusable code, 18
literals, definition, 78–79
logic errors, 60
logical operators, 188–189, 196–197
`long` type, range of values, 145
loops
  constructing, 230–236
  end of file, bypassing, 273–274
  end of file, checking for, 269–271
  ending, 229, 235
  infinite, 235–236
  iterations, 227
  limits, setting, 305–306
  nesting, 265–276
  overview, 226–227
  priming, 237–244
  tracing, 228–229
  Twenty-One, sample program, 230–236
  variable values, 234–235
loops, `do` statements
  deleting files with, 287–291
  description, 291–292
  overview, 286–287
  syntax, 292
loops, enhanced `for` statements
  creating, 293–295
  limitations, 320
  looping through arrays, 319–320
  nesting, 295–299
  overview, 292–293
loops, `for` statements
  conditions, 307–309
  initializing, 281–283
  nesting, 284–286
  overview, 278–280
  sample code, 278
  structure of, 280–281
  versus `while` statements, 283
loops, `while` statements. *See also* `if` statements
  blocks, 230
  versus `if` statements, 230
  versus `for` statements, 283
  syntax, 227

• M •

`main` method
  declaring variables, 353–354
  definition, 77
  description, 85
  multiple, 325
  sample code, 87–88
  single, 323–324
Managing Your Money Online For Dummies, 355
markers, Eclipse IDE, 66
`Math` class, 415
ME (Micro Edition), 27
menu buttons, Eclipse IDE, 66
messages, error
  debugging, 111–112
  `if` statements, 176
  runtime, 111–112
method calls
  description, 83–84
  dot notation, 344, 363
  example, 360–362
  non-static methods, 346
  static methods, 346
method declaration
  within a class, 357–360
  description, 83–84
method headers
  checking for consistent type usage, 377
  definition, 83–84
  description, 359–360
  example, 371–372
methods. *See also* objects; *specific methods*
  ( ) (parentheses), in method names, 363
  abstract, 106–107
  bodies, 83–84, 360
  case sensitivity, 87
  comparing strings, 341–344

methods *(continued)*
definition, 82
`equals`, 346
`findWithinHorizon`, 98, 155–156
flow of control, 362–363
`getCurrencyInstance`, 348–349
`getProperty`, 416
getting values from, 373–377
`main`, 85, 87–88
`next`, 98, 338
`nextDouble`, 98
`nextInt`, 98
`nextLine`, 98, 338
non-static, 345–350
overview, 339–341
`pack`, 384
`parseInt`, 346–348, 414
passing values to, 367–372
punctuation, 363
return values, 373–377
`setDefaultCloseOperation`, 383–384
`setVisible`, 384
static, 345–350, 352–353
`System.out.println`, 86–88
`toUpperCase`, 149–150
methods, `main`
declaring variables, 353–354
definition, 77
description, 85
multiple, 325
sample code, 87–88
single, 323–324
methods, `print`
displaying strings, 338–339
displaying text lines, 119
writing to files, 259–260
methods, `println`
definition, 78
description, 86–87
displaying multiple lines, 119
displaying strings, 338–339
writing to files, 259–260
Micro Edition (ME), 27
minus equal sign (-=), decrement operator, 138–141
minus sign (-), subtraction operator, 133
minus signs (--), decrement operator, 138–141
modulus operator. *See* remainder operator
mortgage program, sample, 48–52
moving variables, 125–126
multiplication operator, 133
multiplying numbers, 133, 143
multithreading, 395

• N •

nesting
enhanced `for` statements, 295–299
`if` statements, 199–202
loops, 265–276
`for` statements, 284–286
news, web links, 407–408, 410–411
newsgroups, web links, 408
`next` method, 98, 338
`nextDouble` method, 98
`nextInt` method, 98
`nextLine` method, 98, 338
No Fluff Just Stuff, web link, 409
non-static methods, 345–350
not-equal operator, 160, 163
`null`, reserved word, 75
null layout, 397
`NumberFormat` class, 348–350, 415
numbers
adding, 133, 143
comparing, 160–163
dividing, 133, 143
expressions, 141–142
formatting, punctuation, 79
formatting as currency, 348–349, 415
incrementing/decrementing, 138–141
multiplying, 133, 143
primitive types, 145–146
reading, 156–157

remainders, 134–136
rounding, 133–134
size, 144–146
subtracting, 133, 143
numbers, converting
to binary, 414
to hexadecimal, 414
from strings, 346–348, 414
to strings, 348–349
type conversion. *See* casting
numbers, decimal. *See also* `double` type; `float` type
comparing, 160–163
definition, 121
reading from the keyboard, 122–124
types, 146
numbers, whole. *See also* `byte` type; `int` type; `long` type; `short` type
definition, 130
reading from the keyboard, 131–133

• O •

object code
definition, 12
vs. source code, 13
objects. *See also* classes; methods
versus classes, 332–333
without classes, 334
creating, 326–332
instances, 329, 334
from multiple classes, 334
overview, 326–329
reference types, 323, 332
referencing parts of, 329
self-filling, 366–367
versus tables, 332–333
offline installation *versus* online, 27
On the Web icon, 6
online installation *versus* offline, 27
online resources
API documentation, 408–409, 413
blogs, 409
Developer.com/Gamelan, 408
The DZone at Javalobby, 409
Free On-Line Dictionary of Computing, 410
FreewareJava.com, 408
The Giant Java Tree, 408
IBM developerWorks, 408
Java Beginner Tutorial, 408
The Java Boutique, 408
Java Shareware, 408
The java.blogs community, 409
The JavaRanch, 408
job opportunities, 409–410
JRoller, 409
The Language List, 410
news, 407–408, 410–411
newsgroups, 408
No Fluff Just Stuff, 409
opinions and advocacy, 409, 410–411
Oracle Blogs, 409
Oracle official site, 407
podcasts, 409
programming languages, 410
projects, 411
The Quine Page, 410
reviews, 407–408, 410–411
Richard Baldwin's website, 408
sample code, 407–408
Slashdot, 410–411
SourceForge, 410–411
Steinar Knutsen's Language list page, 410
for this book, 21
tutorials, 408
OOP (object-oriented programming), 88, 321. *See also* classes; objects
opinions and advocacy, web links, 409, 410–411
Oracle Blogs, web link, 409
Oracle official site, web link, 407
`out` identifier, 77–78
output, 237. *See also* `print` method; `println` method

• P •

`pack` method, 384
packages
  definition, 350
  Eclipse IDE, 55
  fully qualified names, 351
  `java.lang`, 351
  `java.util`, 350–351
  simple names, 351
parentheses (( )), in method names, 363
`parseInt` method, 346–348, 414
passing values to methods, 367–372
percent sign (%), remainder operator, 133
perspectives, Eclipse IDE, 67–68
plus equal sign (+=), increment operator, 138–141
plus sign (+)
  addition operator, 133, 364, 366
  concatenating strings, 364
plus signs (++), increment operator, 138–141
podcasts, web links, 409
portability, Java code, 17–18
postincrement operators, 139–141
predecrement operators, 139, 141
previewing code, Eclipse IDE, 50
priming loops, 237–244
primitive types
  non-numeric, 166
  numeric, 145–146
  versus reference types, 332
`print` method
  displaying strings, 338–339
  displaying text lines, 119
  writing to files, 259–260
`println` method
  definition, 78
  description, 86–87
  displaying multiple lines, 119
  displaying strings, 338–339
  writing to files, 259–260
`PrintStream` class, 252
prioritizing versions, 34–35
problems, solving. *See* debugging; troubleshooting
programmers, definition, 10, 123
programming languages
  description, 11
  web links, 410
programs. *See also* code; Java program elements; *specific programs*
  applets, 413
  definition, 9
  example, 9
projects
  Eclipse IDE, 50, 53–60
  web links to, 411
prompts, definition, 123
punctuation
  { } (curly braces), 79–80, 89
  debugging, 103–109
  methods, 363
  numbers, 79
  spaces, 113

• Q •

question mark colon (? :), conditional operator, 222
The Quine Page, web link, 410

• R •

`Random` class, 179–180
random number generation, 179–180, 358, 365
`.rar` files, 24
reading
  characters *versus* numbers, 156–157
  single characters, 156–157
  strings, 338–339
reading, from files. *See also* disk-oriented programs
  duplicate files, 307
  examples, 258–259

file not found, 252, 255–257
input files, creating, 253
Java code for, 252
template for, 250–251
reading, from the keyboard. *See also* `Scanner` class
decimal numbers, 122–124
echoing keystrokes. *See* `EchoLine` program
`EchoLine` program, 98–99
methods for, 98–99, 122–124
whole numbers, 131–133
red jagged underlines, 58
reference types, 323, 332
remainder operator, 133
remainders, 134–136
Remember icon, 6
repeating instructions. *See* loops
reports, creating from arrays, 304, 305–306, 308, 314–315
reserved words, 75
return values, methods, 373–377
reusing code, 17–18
reviews, web links, 407–408, 410–411
Richard Baldwin's website, web link, 408
rounding, 133–134
running, code
canned Java programs, 48–52
code you've created, 52–61
description, 13, 15–17
in Eclipse IDE, 48–52
mortgage program, sample, 48–52
text-based programs, 48
running, Eclipse IDE for the first time, 38–46
runtime error messages, debugging, 111–112
runtime exceptions, unchecked, 60–61

• S •

sample programs
downloading, 22, 24, 43–46
importing into Eclipse IDE, 43–46
web links, 407–408
`Scanner` class
description, 99, 415
`findWithinHorizon` method, 98
`java.util` package, 350–351
`next` method, 98
`nextDouble` method, 98
`nextInt` method, 98
`nextLine` method, 98
SDK (Software Development Kit). *See* JDK (Java Development Kit)
SE (Standard Edition), 26–27
searching input, 98, 155–156
semicolon (;), ending statements, 86
serialVersionUID, 380
`setDefaultCloseOperation` method, 383–384
`setVisible` method, 384
`short` type, range of values, 145
simple names, 351
simple types
non-numeric, 166
numeric, 145–146
versus reference types, 332
Sindell, Kathleen, 355
64-bit systems *versus* 32-bit, 26
skinning cats, 181
slash (/), division operator, 133
slash asterisk... (/*...*/), traditional comment, 80
slash asterisks (/**...*/), Javadoc comments, 80
slash equal sign (/=), decrement operator, 138–141
Slashdot, web link, 410–411
slashes (//), end-of-line comment, 80
software. *See* programs; tools; *specific software*
Software Development Kit (SDK). *See* JDK (Java Development Kit)
source code
definition, 12
versus object code, 13
SourceForge, web link, 410–411

spaces, punctuating code, 113
spelling, debugging, 103, 109–110
Standard Edition (SE), 26–27
statements, definition, 86. *See also specific statements*
statements, `do`. *See also* loops
  deleting files with, 287–291
  description, 291–292
  overview, 286–287
  syntax, 292
statements, enhanced `for`. *See also* loops
  creating, 293–295
  limitations, 320
  looping through arrays, 319–320
  nesting, 295–299
  overview, 292–293
statements, `for`. *See also* loops
  conditions, 307–309
  initializing, 281–283
  nesting, 284–286
  overview, 278–280
  sample code, 278
  structure of, 280–281
  versus `while` statements, 283
statements, `if`. *See also* `switch` statements; `while` statements
  blocks, 185
  `boolean` variables, 194–196
  cascading, 202–204
  compound statements, 174
  conditions, combining, 189–191, 193–194
  conditions, grouping, 198–199
  description, 173–177
  `else` clauses, 174, 181–183
  enumerating outcomes, 205–208
  error messages, 176
  `if` clauses, 174
  indenting, 179, 181
  initializing variables, 191–192
  logical operators, 188–189, 196–197
  nesting, 199–202
  sample code, 173
  sample programs, 177–179, 183–185
  versus `while` statements, 230
statements, `switch`. *See also* `if` statements
  `break` statements, 212–213, 217–219
  `case` clauses, 212–213
  conditional operators, 221–223
  `default` clause, 213, 216
  description, 213–216
  fall-through, 219–221
  overview, 209–211
  sample code, 209–211
statements, `while`. *See also* `if` statements; loops
  blocks, 230
  versus `if` statements, 230
  versus `for` statements, 283
  syntax, 227
static import, 186
static methods, 345–350, 352–353
Steinar Knutsen's Language list page, web link, 410
`String` class
  definition, 77, 335–336
  description, 416
  reading/writing values, 338–339
  sample code, 336–337
  variables, 337–338
`String` type *versus* `int`, 341–342. *See also* `char` type
strings
  adding `char` type, 366
  assigning `char` type to, 365
  comparing, 341–342
  concatenating, 364
  converting from numbers, 348–349
  converting to numbers, 346–348
  displaying, 338–339
  reading/writing, 338–339
  tokenizing, 416
`StringTokenizer` class, 416
subtracting numbers, 133, 143
subtraction operator, 133

switch statements. *See also* if statements
  break statements, 212–213, 217–219
  case clauses, 212–213
  conditional operators, 221–223
  default clause, 213, 216
  description, 213–216
  fall-through, 219–221
  overview, 209–211
  sample code, 209–211
switching workspaces, 52–53
System class, 351
System identifier, 77
System.out variable, abbreviating, 351

• T •

tab groups, Eclipse IDE, 65
tab stops, 237
tables *versus* classes and objects, 332–333
tabs, Eclipse IDE
  definition, 65
  displaying, 66
.tar files, 24
Technical Stuff icon, 6
temperatures, converting, 160–163
templates, reading/writing data, 250–251
text-based programs, 48
32-bit systems *versus* 64-bit, 26
Tip icon, 6
toolbars, Eclipse IDE, 66
tools. *See also* IDEs (integrated development environments); *specific tools*
  downloading, 22–23, 25
  minimum requirements, 18–19
  on your hard drive, 20
toUpperCase method, 149–150
tracing loops, 228–229
troubleshooting. *See also* debugging
  disk-oriented programs, 255–257
  file problems, 255–257
true, reserved word, 75
true/false conditions. *See* boolean type
tutorials, web links, 408
Twenty-One, sample program, 230–236
types
  byte, 145
  converting. *See* casting
  decimal numbers, 146
  definition, 120–121
  double, 145
  enum type, 205–208
  float, 145
  long, 145
  reference, 323, 332
  short, 145
  String, 341–342
  variables, 117
  whole numbers, 146
types, boolean
  description, 157–159
  range of values, 166
types, char. *See also* String type
  adding to strings, 366
  assigning to strings, 365
  capacity, 151
  definition, 148
  example, 148
  range of values, 166
types, int
  converting from character, 346–348, 414
  converting to binary, 414
  converting to hexadecimal, 414
  minimum/maximum values, finding, 414
  range of values, 145
  versus String, 341–342
types, primitive
  non-numeric, 166
  numeric, 145–146
  versus reference types, 332
types, simple
  non-numeric, 166
  numeric, 145–146
  versus reference types, 332

• U •

unchecked runtime exceptions, 60–61
uncompressing files, 24
underscore (_)
  in class names, 56
  in literals, 78
Unicode, 166–167
uninstalling old versions, 33–35
UNIX For Dummies: Quick Reference, 5th Edition, 355
unzipping files, 24
URLs of interest
  API documentation, 408–409, 413
  blogs, 409
  Developer.com/Gamelan, 408
  The DZone at Javalobby, 409
  Free On-Line Dictionary of Computing, 410
  FreewareJava.com, 408
  The Giant Java Tree, 408
  IBM developerWorks, 408
  Java Beginner Tutorial, 408
  The Java Boutique, 408
  Java Shareware, 408
  The java.blogs community, 409
  The JavaRanch, 408
  job opportunities, 409–410
  JRoller, 409
  The Language List, 410
  news, 407–408, 410–411
  newsgroups, 408
  No Fluff Just Stuff, 409
  opinions and advocacy, 409, 410–411
  Oracle Blogs, 409
  Oracle official site, 407
  podcasts, 409
  programming languages, 410
  projects, 411
  The Quine Page, 410
  reviews, 407–408, 410–411
  Richard Baldwin's website, 408
  sample code, 407–408
  Slashdot, 410–411
  SourceForge, 410–411
  Steinar Knutsen's Language list page, 410
  for this book, 21
  tutorials, 408
users, definition, 123

• V •

values, in arrays, 312–314
values, in variables
  assigning, 118
  definition, 117
  initializing, 124–125
variable declarations
  combining, 126–127, 137
  description, 120–121
variables
  adding, 143
  assignment statements, 118
  declaring, 353–354
  definition, 116
  dividing, 143
  incrementing/decrementing, 143–144
  initializing, 191–192
  moving, 125–126
  multiplying, 143
  names, 118
  reusing, 151–154
  subtracting, 143
  type, 117
  uses for, 116–118
variables, values
  assigning, 118
  definition, 117
  initializing, 124–125
versions
  conflicts, GUI (Graphical User Interface), 380
  installing multiple, 33–35
  Java numbering scheme, 26
vertical bars (||), logical or, 188–189
views, Eclipse IDE
  active, 64
  bringing to the forefront, 64

definition, 64
stacking, 64
toolbars, 66
virtual machine, 12
Visual Basic language, example, 11

• W •

Walheim, Lance, 355
Warning icon, 6
web links
API documentation, 408–409, 413
blogs, 409
Developer.com/Gamelan, 408
The DZone at Javalobby, 409
Free On-Line Dictionary of Computing, 410
FreewareJava.com, 408
The Giant Java Tree, 408
IBM developerWorks, 408
Java Beginner Tutorial, 408
The Java Boutique, 408
Java Shareware, 408
The java.blogs community, 409
The JavaRanch, 408
job opportunities, 409–410
JRoller, 409
The Language List, 410
news, 407–408, 410–411
newsgroups, 408
No Fluff Just Stuff, 409
opinions and advocacy, 409, 410–411
Oracle Blogs, 409
Oracle official site, 407
podcasts, 409
programming languages, 410
projects, 411
The Quine Page, 410
reviews, 407–408, 410–411
Richard Baldwin's website, 408
sample code, 407–408
Slashdot, 410–411
SourceForge, 410–411
Steinar Knutsen's Language list page, 410
for this book, 21
tutorials, 408
`while` statements. *See also* `if` statements; loops
blocks, 230
versus `if` statements, 230
versus `for` statements, 283
syntax, 227
whole numbers. *See also* `byte` type; `int` type; `long` type; `short` type
definition, 130
reading from the keyboard, 131–133
WindowBuilder
description, 385
installing, 385–386
WindowBuilder, GUI classes
bottlenecks, 394–395
creating, 387–388
deadlocks, 394–395
event-dispatching threads, 395
extending, 391–393
instantiating, 393–394
multithreading, 395
running, 388–389
sample code, 389–390
windows. *See also* frames; GUI (Graphical User Interface)
close operations, specifying, 383–384
displaying on a screen, 380–382
Eclipse IDE, 63
visibility, setting, 384
word length
32-bit systems *versus* 64-bit, 26, 28–29
definition, 28
workbench, Eclipse IDE, 62
workspaces
definition, 48
launching, 48–50
separating, 52–53
switching, 52–53
wrapping displayed lines, 119

writing
  code, issuing commands for, 19. *See also* Eclipse IDE
  disk-oriented programs, 257–260
  strings, 338–339
writing, to files. *See also* disk-oriented programs; `PrintStream` class
  examples, 259–260, 262–263
  Java code for, 252
  output file, viewing, 254–255
  template for, 250–251

• *Y* •

yellow jagged underlines, 61
Young, Margaret Levine, 355

• *Z* •

zeros and ones, 119–121
`.zip` files, 24

Apple & Mac

iPad 2 For Dummies,
3rd Edition
978-1-118-17679-5

iPhone 4S For Dummies,
5th Edition
978-1-118-03671-6

iPod touch For Dummies,
3rd Edition
978-1-118-12960-9

Mac OS X Lion
For Dummies
978-1-118-02205-4

Blogging & Social Media

CityVille For Dummies
978-1-118-08337-6

Facebook For Dummies,
4th Edition
978-1-118-09562-1

Mom Blogging
For Dummies
978-1-118-03843-7

Twitter For Dummies,
2nd Edition
978-0-470-76879-2

WordPress For Dummies,
4th Edition
978-1-118-07342-1

Business

Cash Flow For Dummies
978-1-118-01850-7

Investing For Dummies,
6th Edition
978-0-470-90545-6

Job Searching with Social
Media For Dummies
978-0-470-93072-4

QuickBooks 2012
For Dummies
978-1-118-09120-3

Resumes For Dummies,
6th Edition
978-0-470-87361-8

Starting an Etsy Business
For Dummies
978-0-470-93067-0

Cooking & Entertaining

Cooking Basics
For Dummies, 4th Edition
978-0-470-91388-8

Wine For Dummies,
4th Edition
978-0-470-04579-4

Diet & Nutrition

Kettlebells For Dummies
978-0-470-59929-7

Nutrition For Dummies,
5th Edition
978-0-470-93231-5

Restaurant Calorie Counter
For Dummies,
2nd Edition
978-0-470-64405-8

Digital Photography

Digital SLR Cameras &
Photography For Dummies,
4th Edition
978-1-118-14489-3

Digital SLR Settings
& Shortcuts
For Dummies
978-0-470-91763-3

Photoshop Elements 10
For Dummies
978-1-118-10742-3

Gardening

Gardening Basics
For Dummies
978-0-470-03749-2

Vegetable Gardening
For Dummies,
2nd Edition
978-0-470-49870-5

Green/Sustainable

Raising Chickens
For Dummies
978-0-470-46544-8

Green Cleaning
For Dummies
978-0-470-39106-8

Health

Diabetes For Dummies,
3rd Edition
978-0-470-27086-8

Food Allergies
For Dummies
978-0-470-09584-3

Living Gluten-Free
For Dummies,
2nd Edition
978-0-470-58589-4

Hobbies

Beekeeping
For Dummies,
2nd Edition
978-0-470-43065-1

Chess For Dummies,
3rd Edition
978-1-118-01695-4

Drawing For Dummies,
2nd Edition
978-0-470-61842-4

eBay For Dummies,
7th Edition
978-1-118-09806-6

Knitting For Dummies,
2nd Edition
978-0-470-28747-7

**Language &
Foreign Language**

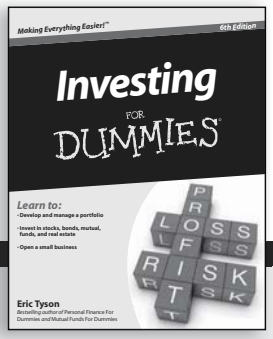
English Grammar
For Dummies,
2nd Edition
978-0-470-54664-2

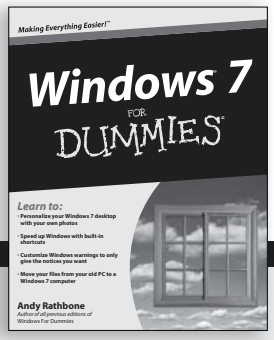
French For Dummies,
2nd Edition
978-1-118-00464-7

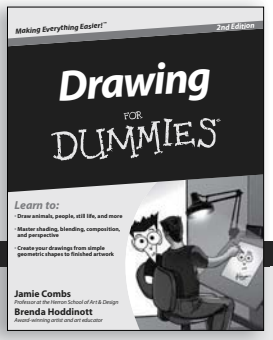
German For Dummies,
2nd Edition
978-0-470-90101-4

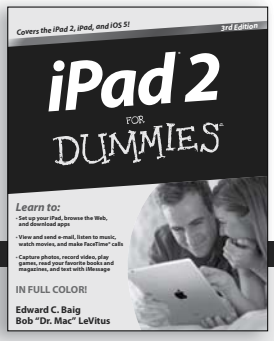
Spanish Essentials
For Dummies
978-0-470-63751-7

Spanish For Dummies,
2nd Edition
978-0-470-87855-2









Available wherever books are sold. For more information or to order direct: U.S. customers visit www.dummies.com or call 1-877-762-2974.
U.K. customers visit www.wileyeurope.com or call (0) 1243 843291. Canadian customers visit www.wiley.ca or call 1-800-567-4797.
Connect with us online at www.facebook.com/fordummies or @fordummies

Math & Science

Algebra I For Dummies, 2nd Edition
978-0-470-55964-2

Biology For Dummies, 2nd Edition
978-0-470-59875-7

Chemistry For Dummies, 2nd Edition
978-1-1180-0730-3

Geometry For Dummies, 2nd Edition
978-0-470-08946-0

Pre-Algebra Essentials For Dummies
978-0-470-61838-7

Microsoft Office

Excel 2010 For Dummies
978-0-470-48953-6

Office 2010 All-in-One For Dummies
978-0-470-49748-7

Office 2011 for Mac For Dummies
978-0-470-87869-9

Word 2010 For Dummies
978-0-470-48772-3

Music

Guitar For Dummies, 2nd Edition
978-0-7645-9904-0

Clarinet For Dummies
978-0-470-58477-4

iPod & iTunes For Dummies, 9th Edition
978-1-118-13060-5

Pets

Cats For Dummies, 2nd Edition
978-0-7645-5275-5

Dogs All-in One For Dummies
978-0470-52978-2

Saltwater Aquariums For Dummies
978-0-470-06805-2

Religion & Inspiration

The Bible For Dummies
978-0-7645-5296-0

Catholicism For Dummies, 2nd Edition
978-1-118-07778-8

Spirituality For Dummies, 2nd Edition
978-0-470-19142-2

Self-Help & Relationships

Happiness For Dummies
978-0-470-28171-0

Overcoming Anxiety For Dummies, 2nd Edition
978-0-470-57441-6

Seniors

Crosswords For Seniors For Dummies
978-0-470-49157-7

iPad 2 For Seniors For Dummies, 3rd Edition
978-1-118-17678-8

Laptops & Tablets For Seniors For Dummies, 2nd Edition
978-1-118-09596-6

Smartphones & Tablets

BlackBerry For Dummies, 5th Edition
978-1-118-10035-6

Droid X2 For Dummies
978-1-118-14864-8

HTC ThunderBolt For Dummies
978-1-118-07601-9

MOTOROLA XOOM For Dummies
978-1-118-08835-7

Sports

Basketball For Dummies, 3rd Edition
978-1-118-07374-2

Football For Dummies, 2nd Edition
978-1-118-01261-1

Golf For Dummies, 4th Edition
978-0-470-88279-5

Test Prep

ACT For Dummies, 5th Edition
978-1-118-01259-8

ASVAB For Dummies, 3rd Edition
978-0-470-63760-9

The GRE Test For Dummies, 7th Edition
978-0-470-00919-2

Police Officer Exam For Dummies
978-0-470-88724-0

Series 7 Exam For Dummies
978-0-470-09932-2

Web Development

HTML, CSS, & XHTML For Dummies, 7th Edition
978-0-470-91659-9

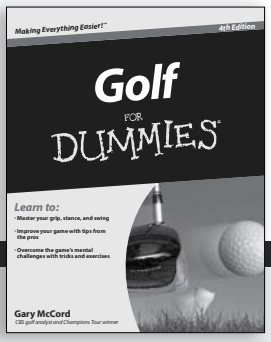
Drupal For Dummies, 2nd Edition
978-1-118-08348-2

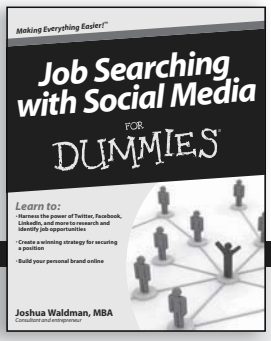
Windows 7

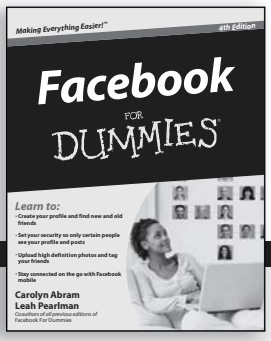
Windows 7 For Dummies
978-0-470-49743-2

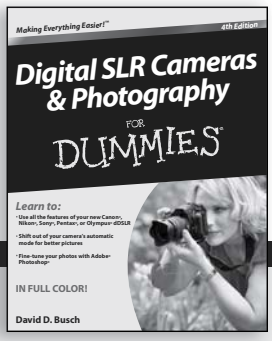
Windows 7 For Dummies, Book + DVD Bundle
978-0-470-52398-8

Windows 7 All-in-One For Dummies
978-0-470-48763-1









Available wherever books are sold. For more information or to order direct: U.S. customers visit www.dummies.com or call 1-877-762-2974.
U.K. customers visit www.wileyeurope.com or call (0) 1243 843291. Canadian customers visit www.wiley.ca or call 1-800-567-4797.
Connect with us online at www.facebook.com/fordummies or @fordummies

Mobile Apps FOR DUMMIES®

There's a Dummies App for This and That

With more than 200 million books in print and over 1,600 unique titles, Dummies is a global leader in how-to information. Now you can get the same great Dummies information in an App. With topics such as Wine, Spanish, Digital Photography, Certification, and more, you'll have instant access to the topics you need to know in a format you can trust.

To get information on all our Dummies apps, visit the following:

www.Dummies.com/go/mobile from your computer.

www.Dummies.com/go/iphone/apps from your phone.